Response: December 11 Edifecs Questions

Contents

Question1: Define Concept Bundles (Namespaces for OnLEP)

Examples:

Edifecs_Demographics_Bundle_V1

Edifecs_Clinical_Bundle_V1

We must have support for these namespaces. This is an absolute MUST.

Lifecycle examples for these name-spaces:

Example 1. : Edifecs_Demographics_Bundle_V2 is an extension of ..._V1 but is a different namespace

- Redefines namespace with Equivalence to older (to share the data)
- New concepts, updates and deletions and extensions

Example 2. : Edifecs_Demographics_Bundle_V1.1

- Shares the older name-space with additions only (non sealed extensions)
- No extension of sealed concepts (same meaning as 'scala' sealed traits)
- No updates
- No deletes

Questions:

- Can the above be supported?
- Any way to migrate/copy data associate with one to the other?
- Especially for updates (namespace change from ..._V1 to ..._V2 and some changes)
- Any ways to hook migration scripts?

Q1 ANSWER: Namespaces

We have limited support for namespaces today.

However,

1. Namespaces are currently supported in the OnLEP metadata, although there is no concept of inheritance or mixin to suggest that namespace content from one namespace can be subsumed/inherited by another namespace. If you add new Namespaces, they must be supported by the PMML compiler.
2. In terms of namespace usage in the models, the namespaces must currently be either System or

PMML. They are automatically considered for type and function lookups by the PMML Compiler, with System searched first followed by PMML.

In a post 1.0 sprint, it is expected that richer semantics for namespaces will be provided with the following extensions:

1. Namespace search list will be added to model data dictionary to be used to control type and UDF lookups. Used to locate the data types and UDFs found in the model, the first match is returned. This suggests that a newer implementation of some behavior can take precedence over the older behavior of the same name simply by ordering the newer namespace in front of the prior one.
2. Explicit use of a '.' qualified namespace will be supported on the type and UDF uses in the fields and derived fields.
3. Related to item 2, specifically referring to another model's value of some derived field (i.e., a derived concept) by mentioning the model qualified derived field name from that other model in this model will be possible. In this way the "new" version of the concept can use the older version of the concept as an argument (a parameter) to the function responsible for computing the concept in the new way.

For Example 1: *Edifecs_Demographics_Bundle_V2 is an extension of ..._V1 but is a different namespace*

1. **New concepts and updates** (by this I am assuming a new replacement version) to existing ones previously defined in the prior namespace are easily handled by what is described in the previous point 1.
2. **Extensions**, however, suggest some sort of mixin strategy where more than one object in the namesearchspace contains the required behavior, or is it just a simply a matter of the values that are now known to represent the concept?
3. If this is **behavioral**, not just data values, this requirement means derived fields must either inherit from another (IsA) or contain another object (HasA) and its values. This doesn't map to the PMML very well – which has no behavioral inheritance. PMML is a functional language. I believe this notion is asking for object oriented features, which I suspect maps more closely to the OWL class and its hierarchy.
4. If, on the other hand, this is a **value only** issue where the new concept HAS the older concept value, then the new concept could be represented as a structure that has an instance of the older concept value in it. That is one possibility.
5. Another possibility is that the **derived field** (i.e., the concept from the prior model**) should execute on the data event** (the message) and compute its value that in turn is used by the new derived

and related concept that consumes the same message plus the calculation of the prior model containing the prior generation concept.

6. This is the design for the DAG (Directed Acyclic Graph) and is in the current code. The emitted (supplementary or predicted) fields of a model can be referenced by other models. This relationship is captured at compile time by the PMMLCompiler and recorded in the model metadata of the consuming model. The engine then prepares a partial ordering of these models (i.e., a DAG) and all others that are part of the engine's model working set to make sure that the inputs for any given model are available before it executes.

The diagram below shows a simplified version of this interaction. The inpatient data *for a specific beneficiary,* can be used in Model 4, then both the original inpatient data and the Model 4 result can be used in Model 9, and the original inpatient data and Model 9 result can be used in Model 45.

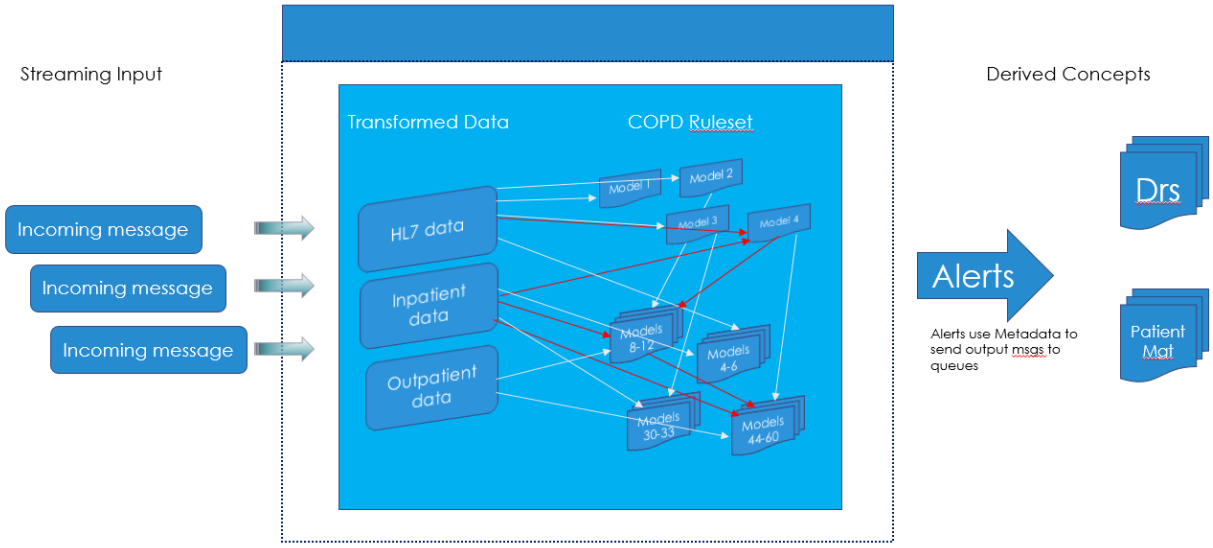

Figure 1 - Model Dependency Relationships

7. The **deletion of a concept** – our suggestion might be that the model containing some derived concept is simply ignored by the new derived concept computed in a later model. We should work through the impact of an actual deletion of a concept.
8. Could you supply a set of small pedagogical examples (or cite references) of OWL that use/reflect these notions: new concepts, updates and deletions and extensions?

For example 2: *Edifecs_Demographics_Bundle_V1.1*

1. The namesearchpath would manage this for the non-sealed *namespaces*, if I understand. The older implementations would handle certain concepts while the new concepts in the "new" namespace can either add new concepts or provide a new implementation of an older concept from the previous version of the namespace.
2. The sealed idea, are you asking for semantics (metadata support?) to support the idea of a sealed concept?

Observations and conclusions.

1. The use of the namespace maps to the use of the model name as either the namespace or part of the namespace name in some respects.
2. I think that the older namespace (i.e., model name) being referenced in the newer namespace (new model) as perhaps an argument to the new computation is doable.-I suppose the question is, "is that what you are requesting?" To be a bit more concrete, consider the following snippet of PMML that is using a previous concept derivation (found in older model that has EdDemo_V1 as its namespace) from the implementation of the concept in the new derived field (found in a different model with a namespace EdDemoV2). The model names in both cases are the same, 'foobar' The concept name is 'foo':

```
<DerivedField name="foo" dataType="EdDemoV2 .fooBundleType" >
        <Apply function="barComputation">
                <FieldRef field="EdDemoV1.foobar.foo"/>
                <FieldRef field="AATDeficiency"/>
                <FieldRef field="WithEnvironmentalExposures"/>
        </Apply>
</DerivedField>
```

Code snippet 1

3. In the example, the older concept plays a part in the computation of the new example. In addition to the new requirements that there be an AAT deficiency with environment exposure history, the EdDemoV1.foobar.foo computation is reused. Note that the way I am thinking about this, the EdDemoV1.foobar model is part of the current OnLEP engine working set and was just computed on the incoming message (plus the history of the visits and computations on those visits).
4. If this is what you have in mind, we will try to give you an idea of what work items and features will need to be built to satisfy it as not all of this is currently in place in our code base.

Question 2. Define Concepts in Bundles

A rather simplified example is this:

Patient

- patientId
- DOB
- visits Visit
- date
- chronic_cough
- chest_pain

- Visits have history. Every visit that comes in will have data that needs to be stored in history.
- Visit data may be updated (Corrected).
- Patient data may also be corrected (merge of 2 patients as one or split some visits to another patient record).
- Visit data may also be corrected. For example, it may be necessary to correct data if:
    - Someone accidentally put 'chronic_cough' to an older visit (split).
    - Someone accidentally entered observations from 1 visit in 2 visit records and this needs to be merged.

Questions:

- How are these corrections handled from a data storage perspective?
- How are the splits and merges handled from a data storage perspective?
- Is there a way to hook migration scripts? This really is completely automatable btw.
- How do we trigger re-execution of rules - based on these changes/corrections?
- This is really a big issue. Move records over and decide which models/rules to re-execute.
- We cannot go into production without this.

Q2 ANSWER: Concepts in Bundles

Bundled Concepts or Concepts for multiple (single-visit) data points over time:

Concepts are already modeled as fields in our current system. There is, under the covers, very little difference between a concept and a field that refers to ANY type, be it a scalar, a collection of scalar, a container (struct or mapped) or array of container.

The difference is that concepts are tracked explicitly (a Map[ConceptName,Concept]) as well as possibly being part of another Container.

Current capabilities support history based models. Take a look at the COPD model in the trunk/SampleApplication/Medical portion of the tree, especially the message definitions in the MessageAndContainers subdirectory (note that there are versions for both fixed and mapped representations of the historical records). The beneficiary.json contains arrays of inpatient, outpatient, and hl7 messages that are filled by the engine when the message is presented to the models.

History

Models can be history based or interpreted without history. The incoming data can be just from one visit or over multiple years of visits, with no limits except drive and processor space. As an example, let's establish how to use the model to have two years of data in the model at any given time.

Data Storage

In terms of data storage, the engine current handles this. Messages come in and are stored by primary key in data storage (we are using Cassandra and Hbase).

Corrections, Splits and Merges and other fun things

In terms of corrections (updates and deletions to prior messages), these would need to be marked as such ("this is an update") and two keys (**BeneficiaryID** and **ClaimID**) would need to match. In this case, the requirement is that these two keys and the date are part of the update message.

If a model were written to perform the update, it would likely access the change through the EnvContext. There may be other ways to do this, even using the engine to perform simple updates without models. This will require some additional metadata to represent semantics.

Assume we are dealing with two years of data, containing many multiple beneficiary visits, partial or complete. The complete history of Messages are saved based on **Partition key** and **Primary key**. The **Partition** key is used to gather all messages containing the same key together. The **Primary** key is used to uniquely identify each message. When the engine gets the message, it checks whether any messages with the same primary key already exist or not. If found, the engine takes the first occurring message and merges new data into it. If the primary key is not found in the current working set, a new message is added.

Please see the diagram below for merge handling over time.

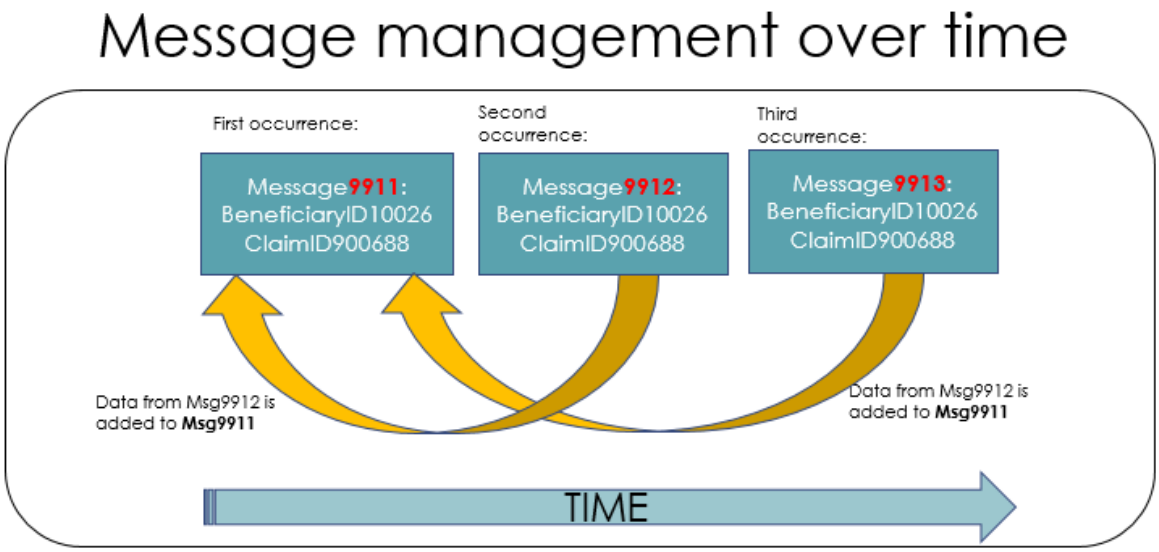

Figure 2: Message Management over Time

Migration Scripts

What are you trying to accomplish with migration scripts? The engine is designed to - and current code does - pull in historical data for any time frame you specifiy. Please see Code Snippet 6 to see how we are accomplishing this. If there is some other functionality you need in "migration", we would like to find out more

Rule Re-execution

For rule re-execution, I think that all affected models that have had their input fields touched in some fashion would be dependent on these update oriented models. Currently, models are executed (which may mean) re-executed whenever an incoming message containing a **Primary key** and containing either partial or full data is passed into the engine.

Question 3. Messages

Buildling on the example from Question 2 above, a simple message is this:

visit
- date

- chronic_cough
- chest_pain
- patient
- patientId

- Now a valid use case for us is for the same visit, there could be more than one message.
  - Message 1 with "chronic_cough" values and missing "chest_pain" values
  - Message 2 with "chest_pain" value with or without "chronic_cough" values

Questions:

How are these partial messages and re-executions handled?
The important thing is re-execution of rules.
With Message 1, there was only a partial set needed by the rule.
With Message 2, the contents are complete and the rule can fire
Realize that some of it is how we author the rules.
But the notion of past-history vs current is what we are focusing on here.
Need a proper technical solution outline for this.

Q3 Answers: Messages

Please see “Rule re-execution” in question 2.

Multiple messages for the same patient for the same visit are currently processed one message at a time. That is, the first message for a given patient is processed with that patient's history for the lookback period. A prediction is made (or not) based upon the ruleset (e.g., COPD-ness). When the second message is received, the same thing happens, but this time the first message is part of the history. Data from the second message is merged into the first message. *Please see Figure 2.*

A prediction could be made on both. Perhaps the first message causes a “low-risk” COPD prediction, while the second predicts “high risk”. It is also conceivable that the first message predicts “at risk” but with additional data from the second prediction the conclusion is that there is “no risk” after all.

Consider that there are two nurses that are working on aspects of the patient's care. Nurse Betty is very efficient and completes her work and posts it to the system which forwards it to Edifecs for processing. Nurse Roger is not as efficient and lets his work stack up, reporting the patient finding several days later. Can the order of the reporting actually change the outcome when all data is finally considered?

Answer: Yes, it can. If Nurse Roger later provides “wrong” information in the same exact field that Nurse Betty filled in with “right” information, it will become the current record. The solution is to write a rule that looks for this and add it to the model.

To some extent this could be mitigated by how the model is coded, as you suggest in the question. For example, let's assume that the Nurse Betty/Nurse Roger example where there is a significant lag between the times that Edifecs receives the events. Let's assume that the messages that Nurse Betty generated produce some alert that is sent to the medical provider for consideration. It would be prudent in every case to forward with that prediction the latest result from the message.-A discerning doctor’s office would see that not all data was considered in the prediction that came from his/her office.-This would likely make the doctor instruct his/her staff to make sure that the report of findings/billing went out together.

Every model might be designed to always report the findings that are germane to the prediction from the very latest visit id or visit date. The human process would need to ensure that in fact all of the patient evidence is present in the alert's supplementary information.

The full history of pertinent facts from the lookback period could be included directly in the model output (e.g., projections of the date/icd9 codes pairs in our COPD CMS data based model from the inpatient and outpatient histories) as evidence of this prediction.

Question 4. Functions

Consider the following function:

hasHistory(msg2.visits.chronic_cough, 2 years before from visit.date)
Note: the first argument is the chronic cough concept - not a value from a Message.

Questions:

How will we define this (the challenge being accessing history and accessing a named meta-data?
Or better still - is this a function you can add to the core?

Q4 Answer: Functions

If the business goal is to see two years of history and evaluate the chronic cough concept, the engine does this now. Simply define your lookback period – see Code Snippet 4. In the actual model

(COPDv1.XMl) see lines 231-245 for time period, and lines 422-461 for chronic cough data.

As mentioned in the Messages section previously, the current system can present the history of “visits”. In our CMS reference models, inpatient, outpatient and hl7 histories are presented with the message as arrays. Here is the end of the beneficiary (~ Patient in your question):

```
{
            "Message": {
            "Name": "Inpatient_Claims",
            "Type": "System.ArrayBufferOfInpatientClaim"
            }
--},
--{
-           "Message": {
            "Name": "Outpatient_Claims",
            "Type": "System.ArrayBufferOfOutpatientClaim"
-           }
--},
--{
-           "Message": {
            "Name": "HL7Messages",
            "Type": "System.ArrayBufferOfHL7"
-           }
--}
- ],
- "PartitionKey": [
--"Desynpuf_Id"
- ]
  }
}
```

Code snippet 2

The partition key is the beneficiary id (field Desynpuf_Id) The inpatient claims and the others are presented as arrays of their type. For field Inpatient_Claims, the array contains instances of InpatientClaim; for Outpatient_Claims, the array has OutpatientClaim instances; for HL7Messages, the members are HL7.-When those types are ingested into the MetadataAPI service, arrays, arraybuffers and

others are automatically created.

Basically, your Patient message's visits would look like this (other fields left out):

```
{--.
--.
--.
                    {
-           "Message": {
            "Name": "visits",
            "Type": "System.ArrayBufferOfVisit"
-           }
- ],
- "PartitionKey": [
--"patientId"
- ]
  }
```

Code snippet 3

To answer your questions, I am assuming that your HasHistory function is intended to determine if the chronic_cough concept exists in any of the Visit messages from the visits history for the look back period described in the second argument., count them, and answer true if the count is greater than zero.

Visits would be represented in your Patient message as an ArrayOfVisit. The chronic_cough concept is a field in the visit. This concept's type could be a Container (a simple structure with known fields or a mapped representation that is useful for sparse data representations ... lots of possible fields but only a few actually present in an instance). Its type could be a scalar or a boolean for that matter.

Either implementation should use a “filter” function that will build a subset of the Visit elements in the visits field that have the chronic_cough concept. Let's run through two ways you might approach this. The “generated” PMML is shown. There certainly are other options between these two extremes that could be used. After presentation, some pro/con analysis will be offered:

Possible Implementation #1:

The projection of the concept field is generated/coded inside the HasHistory UDF, a count is obtained of the chronic_cough concepts present, and a 'true' is returned if more than 0 of them are present:

```
<DerivedField name="HasHistoryOfChronicCough" dataType="boolean">
                <Apply function="HasHistory">
                <FieldRef field="msg.visits"/>
                        <Constant type="string">chronic_cough</Constant>
                        <FieldRef field="LookBackPeriodInYears"/>
                </Apply>
</Apply>
```

Code snippet 4

The msg.visits array is passed to the UDF along with the name of the chronic_cough code field, namely "chronic_cough". The LookBackiPeriodInYears is either a derived variable obtained from the EnvContext (access to external configuration table info is possible).

Since visits could have hundreds of possible conditions of note, the chosen representation for Visit is either a message or container that uses the so-called "mapped" style, useful for sparse data situations like this one. The HasHistory UDF might look like this:

```
        def HasHistory( visits : ArrayBuffer[Visit]
                        , conceptName : String
                        , lookBackInYrs : Int) : Boolean = {
                visits.filter(visit => {
                        visit.getOrElseNull(conceptName) != null).size > 0
                })
        }
```

Code snippet 5

Possible Implementation #2:

The visits in the lookback period are computed in the PMML Model instead. The visit date from the visit event that precipitated the model being executed is needed. It is the last Visit in the Patient's Visit array. Here is the PMML representation of how to get that value. Obviously some of this could be pushed into a UDF:

```
<DerivedField name="LastVisit" dataType="Visit">
        <Apply function="Last">
                <FieldRef field="msg.visits"/>
        </Apply>
</DerivedField>
```

```
<DerivedField name="EarliestVisitDate" dataType="integer">
                <Apply function="AsCompressedDate">
                        <Apply function="BeginOfLookBack">
                                <FieldRef field="LastVisit.date"/>
                                <FieldRef field="LookBackPeriodInYears"/>
                        </Apply>
                </Apply>
</DerivedField>

<DerivedField name="VisitsInLookBackPeriod" dataType="ArrayOfVisit”>
                <Apply function="ToArray">
                        <Apply function="ContainerFilter">
                --      <FieldRef field="msg.visits"/>
                                <Constant dataType="fIdent">Between</Constant>
                                <Constant dataType="ident">date</Constant>
                                <FieldRef field="EarliestVisitDate"/>
                                <FieldRef field="LastVisit.date"/>
                                <Constant dataType="boolean">true</Constant>
                        </Apply>
                </Apply>
</DerivedField>

<DerivedField name="HasHistoryOfChronicCough" dataType="boolean">
                <Apply function="greaterThan">
                        <Apply function="CollectionLength">
                                <Apply function="ContainerFilter">
                                        <FieldRef field="VisitsInLookBackPeriod"/>
                                        <Constant dataType="fIdent">Contains</Constant>
                                        <Constant dataType="ident">chronic_cough</Constant>
                                </Apply>
                        </Apply>
                        <Constant dataType="integer">0</Constant>
                </Apply>
</DerivedField>
```

Code snippet 6

Possible Implementation #1 seems better because it is so short.-The downside is these models will look at the visits array again and again for various purposes in order to produce the correct prediction. This suggests that it would be a good idea to cache the visits in the lookback range so they don't have to be filtered with every access.

It is possible to create a caching mechanism in the UDF library that uses the model variables to contain the cached values. However, one would have to initialize the model variable to something that could be detected in the HasHistory function before HasHistory was first called. HasHistory would change then to load the value of the visit variable and decide if it needed to load it before running the filter.

Another option is to consider the UDF lib as a singleton shared by all models running in the working set. This means the model variable would not be used to keep the state of the visits in the lookback period, but some state object would be instantiated that was known only to the running model instance.

Possible Implementation 2, while looking kind of gnarly, is really all boilerplate except for the HasHistoryOfChronicCough derived field. Perhaps we generate PMML for all of the derived fields except that one and then use the HasHistory function similar to Implementation 1. For example,

```
<DerivedField name="HasHistoryOfChronicCough" dataType="boolean">
                    <Apply function="HasHistory">
                    <FieldRef field="VisitsInLookBackPeriod"/>
                              <Constant type="string">chronic_cough</Constant>
                              <FieldRef field="LookBackPeriodInYears"/>
                    </Apply>
</Apply>
```

Code snippet 7

If it is required to build the cache of the visits in the lookback period at the UDF level, let us know. It is possible, but there will be some things in the PMML that will have to be generated or made part of your template, and there may be some initialization that will be needed to be done either by you or perhaps as part of the model generation by the PMML compiler, and you will need to understand how to update the variables from the UDF with a core UDF that we use internally.

Question 5. Runtime: Packaging, Deployment, Metrics and logging and scaling info

- Need documentation on what metrics are reported - and how:
- Can you report to zabbix?
- Need to ensure you are following the 12-factor app guidelines.
- This is needed to run in a PaaS environment

Q5 ANSWERS: Packaging, Deployment, Metrics, and logging and scaling info

Metrics- Zabbix

At this time, we do not support any metrics, or Zabbix.

Platform as a Service (PaaS)

Below are some of the features that can be included with a PaaS offering. How does OnLEP stack up?

- Operating system
    - Linux or *nix. With some effort the platform could possibly be made to run on Windows. It is anticipated that this effort, if undertaken, will be from a partner/contributor to OnLEP as open source. Of course, it can currently run on Windows for testing and development, but in production there are still issues to be resolved with Windows.
- Server-side scripting environment
    - The plan is to use open source installation (as well as start, stop, etc) server scripts. We have temporary code doing these jobs now.
- Database management system
    - Currently, the OnLEP solution has interfaces for Cassandra, HBase, Redis, MapDb
- Simple to add additional implementations for the standard interfaces
    - Nothing precludes access to commercial databases, Spark compute clusters, et al with direct access to these services by the models via the EnvContext.
- Server Software
    - The OnLEP platform is currently written in Scala

Support

OnLEP is open source software. OnLEP partners and unaffiliated organizations will build rich sets of tools and applications around the platform.

Storage

Any storage that supports Linux will work with OnLEP.

Network access

- TCP/IP communication to/between/from the OnLEP cluster nodes
- Tools for design and development
- Scala , sbt, and any of the IDE frameworks (Eclipse, NetBeans, IntelliJ, etc)
- MetadataAPI service
- MetadataAPI console tool for developers
- Metadata extraction tool for User Defined Functions (UDFs)
- Test data conversion tool from csv/spreadsheet to JSON
- sbt Jar dependency script that provides configuration information for the UDF Metadata extraction tool et al

Hosting

No Hosting is anticipated to be provided by LigaData. This sort of service will be supplied by the customer and their cloud vendor. Deployers of OnLEP will need to either host themselves or use a hosting provider.

12 Factors

According to http://12factor.net/, the twelve factors are found in the list below. How does OnLEP stack up?

1. Codebase: One codebase tracked in revision control, many deploys.
   - OnLEP currently uses git and github for its repo mechanism.
   - The platform is factored into discreet components.
   - OnLEP is metadata driven and written to interfaces. This offers fine control over the platform/application division. The interfaces for input adapters, the environmental context, storage interface that is exposed to the models, and use of industry standard queuing mechanisms give the application developer great flexibility.
2. Dependencies: Explicitly declare and isolate dependencies
   - The Simple Build Tool (sbt) provides the package dependency management for OnLEP
3. Config: Store config in the environment
   - OnLEP is totally metadata driven. Metadata is housed in a kv store and made available to all of the components requiring it, including the OnLEP cluster node engines, MetadataAPI service, Message and Container compiler, and the PMML Compiler.

4. Backing Services: Treat backing services as attached resources.
    - All kv stores support a standard storage interface.
    - All input messages are ingested by the OnLEP cluster node via a standard input adapter (defined as a Scala trait). Adapter implementations are configured with the OnLEP configuration file.
    - Similarly, there is an output adapter interface. The standard implementation can be replaced if desired. Like other component trait implementations in this list, the implementation can be selected via the OnLEP configuration file.
    - The EnvContext exposed to all PMML compiled models provides a consistent interface to access external services. Since it is a Scala trait, alternate implementations can be added to support additional functionality needed on a given cluster. This implementation in use is configurable via the OnLEP configuration file used to initialize the OnLEP cluster at startup.
5. Build, release, run: Strictly separate build and run stages
    - GitHub is used to house source code, configurations, and documentation artifacts. It possesses tagging feature so that discrete sets of components can be maintained and managed.
    - The release tools are simple. Multi-cluster setup builds the components, bundles them in an implementation directory, and ships to the target cluster nodes. The nodes all possess the same directory structure. Simple ssh based commands can be used to start the components across all participating nodes.
    - Adapters information, Nodes information and some other cluster configurations are saved in metadata configuration while setting up the multi-cluster setup. *(Please see the Config file at …\|trunk\|SampleApplication\|Medical\|Configs\|EngineConfig.json).*
6. Processes: Execute the app as one or more stateless processes
    - The OnLEP cluster uses an “exactly once” message processing service. If message processing requires prior message history it is retrieved from a Backing resource (typically Cassandra or HBase, but others are possible)
    - All message processing is under transaction.
7. Port binding: Export services via port binding
    - MetadataService as Webservice uses port (changeable in the OnLEP configuration file)
    - OnLEP cluster admin uses port (changeable in the configuration file)
    - Standard Port usage for Kafka (if used), customizable as always
    - Standard Port usage for Zookeeper, customizable as always
    - Standard Ports used for any other storage components, customizable as always
8. Concurrency: Scale out via the process model
    - OnLEP is a cluster based platform that partitions work according to partition keys defined in metadata. It horizontally scales.

- Each cluster node maps to a single process. Multiple nodes may be defined on the same physical box if desired. *(Please see the Config file at …\|trunk\|SampleApplication\|Medical\|Configs\|EngineConfig.json).* (ex: "NodeIpAddr": "localhost" just needs to be changed to the target machine name).
- Each cluster node is multi-threaded.
- Each message is processed under transaction and “Exactly Once”.

9. Disposability: Maximize robustness with fast startup and graceful shutdown
   - Cluster will rebalance the workload (processing partitions) when node changes (either when a new nodes comes to life or existing nodes go down) and partition changes (partitions changes to existing adapters) happened. We have implemented Leaderlatch to manage this.
   - Messages, Containers, Concepts, Functions, Types and Models can be added, and activated/deactivated while the OnLEP engine is running. Zookeeper is the current broadcast mechanism (between Metadata.Api & engine) whenever metadata changes happened.
   - Zookeeper is also used for electing the leader in the cluster. Leader is responsible for re-balancing the work load of the cluster.
10. Development and production parity: Keep development, staging, and production as similar as possible.
    - Models may be added or removed with simple admin command, making quick fixes a simple matter.
    - The OnLEP system is designed to run on top of a number of technologies (backing services, queuing mechanisms. Given the open source nature of the software in use, it is no burden to keep the development, test and production systems quite close in terms of the technology configurations deployed.
    - OnLEP is agnostic with regard to who performs what roles in terms of development, testing and deployment. All three roles can certainly be performed by the same (or different) personnel as desired.
11. Logs: Treat logs as event streams
    - Logs are uniformly managed by the log4j logging component. The applications log statements have been tuned to use the various severity levels, allowing log4j via the runtime configuration to control log verbosity and detail.
    - The log4j allows for configuration of log consumer. Write to a file, write it to a console, write it to an application that forwards it somewhere for further processing. Configure it as you will. Integrate with log query environments is possible.
12. Admin processes: Run admin/management tasks as one-off processes
    - OnLEP currently does not use the Scala REPL to manage its cluster.
    - At this time, the admin commands are driven by Metadata.API. To add a model to the working set, modify the Metadata using the Metadata.API. Then the existing scripts will add it

to the working set of models when engine runs.

- Admin Commands contemplated:
    - add/remove, activate/deactivate model to/from working set
    - add/remove node to/from cluster. Currently initializing nodes in config script. Future version will support this functionatliy.

Thank you very much for the opportunity to engage in this discussion. We look forward to continuing that discussion either over email, on calls, or in person.

The OnLEP development team – particularly Rich Johnson, Pokuri Sreenivasalu, and Ramana Mandava.