Big Data Software in Particle Physics

Jim Pivarski

Princeton University – DIANA-HEP

August 2, 2018
Sometimes considered as a vacuous question, like “What color is your hammer?”
What software should you use in your analysis?

Sometimes considered as a vacuous question, like “What color is your hammer?”

Anything to get the job done!
What software should you use in your analysis?

Sometimes considered as a vacuous question, like “What color is your hammer?”

Anything to get the job done!

But there are real differences and sometimes it matters:
What software should you use in your analysis?

Sometimes considered as a vacuous question, like “What color is your hammer?”

Anything to get the job done!

But there are real differences and sometimes it matters:

▶ Tool for the wrong problem (hammer vs. screwdriver)
What software should you use in your analysis?

Sometimes considered as a vacuous question, like “What color is your hammer?”

Anything to get the job done!

But there are real differences and sometimes it matters:

- Tool for the wrong problem (hammer vs. screwdriver)
- Ease of use/data analyst productivity (ergonomics of handle)
What software should you use in your analysis?

Sometimes considered as a vacuous question, like “What color is your hammer?”

Anything to get the job done!

But there are real differences and sometimes it matters:

- Tool for the wrong problem (hammer vs. screwdriver)
- Ease of use/data analyst productivity (ergonomics of handle)
- Computational performance (head weight)
Why this matters now:

we’re not the only ones analyzing big datasets anymore: “web scale analytics”
We measure globally distributed data in hundreds of PB.

CERN Data Centre passes the 200-petabyte milestone

by Melissa Gaillard

Posted by Stefania Pandolfi on 6 Jul 2017
Voir en français
This content is archived on the CERN Document Server.

CERN Updates
Next stop: the superconducting magnets of the future
21 Sep 2017

CERN openlab tackles ICT challenges of High-Luminosity LHC
21 Sep 2017

Detectors: unique superconducting magnets
20 Sep 2017

CERN’s Data Centre (Image: Robert Hradil, Monika Majer/ProStudio22.ch)
But for “web scale” companies, 100 PB = 1 truck

CERN Data Centre passes the 200-petabyte milestone

by Melissa Gaillard
Their tools are new, but widely used (many eyes on the code)
Their ecosystem was mostly developed independently from ours.
Ideally, we should use the strengths of both

**What we can find off the shelf**
- distributed processing, scale-out
- C++/Python interoperability
- special functions, matrix math
- fitting/minimization, integration, differentiation, interpolation
- symbolic algebra
- advanced statistics
- machine learning
- graphics, advanced plotting
- graphical interfaces, user workflows

**What we still must develop in-house**
- reading/writing ROOT files
- collaboration frameworks, triggers, event reconstruction
- advanced histogramming and fitting
- efficient variable-length lists (our kind of non-relational data)
- domain-specific functions
Distributed processing: Spark

Functional programming, managed data

- Big, distributed dataset is a local variable:

```python
lookup = spark.textFile("lookup.txt").map(int)
table = spark.cassandraTable("T").filter("col > 0")
both = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
```
Distributed processing: Spark

Functional programming, managed data

- Big, distributed dataset is a local variable:

  ```python
  lookup = spark.textFile("lookup.txt").map(int)
table = spark.cassandraTable("T").filter("col > 0")
both = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
  ```

- Distributed job starts when you ask for its output

http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html
Functional programming, managed data

- Big, distributed dataset is a local variable:

  ```python
  lookup = spark.textFile("lookup.txt").map(int)
table = spark.cassandraTable("T").filter("col > 0")
both = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
  ```

- Distributed job starts when you ask for its output
- Spark manages files, task dependencies, retry-on-failure

[http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html](http://www.cs.sfu.ca/CourseCentral/732/ggbaker/content/spark.html)
Distributed processing: Spark

Functional programming, managed data

- Big, distributed dataset is a local variable:

  ```python
  lookup = spark.textFile("lookup.txt").map(int)
table = spark.cassandraTable("T").filter("col > 0")
both = table.join(lookup).where("n == N")
cached = both.map("x**2 + y**2").cache()
hist = cached.reduce(histogram)
cached.saveAsTextFile("tmp.txt")
  ```

- Distributed job starts when you ask for its output
- Spark manages files, task dependencies, retry-on-failure
- Distributed dataset may be in RAM (fast), disk (big), or both (spill-over)
Accessing particle physics data in Spark

**Spark-ROOT**: presents a large set of ROOT files as a Spark DataFrame

```python
mydata = (spark.read.format("org.dianahep.sparkroot")
  .option("tree", "Events")
  .load("many-files/*.root"))
```

https://github.com/diana-hep/spark-root

**XRootD-HDFS**: presents an XRootD service (e.g. EOS) as HDFS for Spark

```python
...load("hdfs://eos.cern.ch/many-files/*.root")
```

https://github.com/opensciencegrid/xrootd-hdfs
Spark-ROOT: presents a large set of ROOT files as a Spark DataFrame

```python
mydata = (spark.read.format("org.dianahep.sparkroot")
          .option("tree", "Events")
          .load("many-files/*.root"))
```

https://github.com/diana-hep/spark-root

XRootD-HDFS: presents an XRootD service (e.g. EOS) as HDFS for Spark

```python
...load("hdfs://eos.cern.ch/many-files/*.root")
```

https://github.com/opensciencegrid/xrootd-hdfs

ROOT’s new RDataFrame adds a Spark-like interface to local ROOT files; potential for driving Spark from ROOT in the future.
Awkwardness of Spark

Spark runs on Java Virtual Machines (ubiquitous in business).
Awkwardness of Spark

Spark runs on Java Virtual Machines (ubiquitous in business).

Hard to interface C++ (especially ROOT) with Java; PySpark performance is limited by its Java-Python tunnel.
Spark runs on Java Virtual Machines (ubiquitous in business).

Hard to interface C++ (especially ROOT) with Java; PySpark performance is limited by its Java-Python tunnel.

Python itself is much more open to interoperability with C++, has similar distributed computing projects: Dask, Joblib, Parsl...
Data analysis ecosystem has grown around Python

Python’s Scientific Stack
Data analysis ecosystem has grown around Python

Python’s Scientific Stack

The Unexpected Effectiveness of Python in Science
Jake VanderPlas @jakevdp PyCon 2017
Data analysis ecosystem has grown around Python

Python’s Scientific Stack

- matplotlib
- pandas
- Bokeh
- xarray
- NumPy
- jupyter
- SciPy
- IPython
- DASK
- Numba

The Unexpected Effectiveness of Python in Science
Jake VanderPlas@jakevdp
PyCon 2017
Data analysis ecosystem has grown around Python

Python’s Scientific Stack

- StatsModels
- SymPy
- NetworkX
- scikit-image
- matplotlib
- pandas
- Bokeh
- xarray
- NumPy
- jupyter
- SciPy
- PyMC
- DASK
- Numba
Data analysis ecosystem has grown around Python

(and many, many more)
Particularly machine learning
In some fields, adoption is astronomical...
PyROOT: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.
**PyROOT:** general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

**root_numpy:** good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.
**PyROOT**: general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

**root_numpy**: good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

**uproot**: reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

```
bytes of ROOT file → C++ objects → Numpy arrays
```
Accessing ROOT data in Python

**PyROOT:** general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

**root_numpy:** good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

**uproot:** reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

白银 of ROOT file $\rightarrow$ C++ objects $\rightarrow$ Numpy arrays
**PyROOT:** general solution, but slow if called in a loop over events; new ROOT features like `TTree::AsMatrix()` fill Numpy arrays directly, but only for primitive types.

**root_numpy:** good solution for many years, except slow for nested data (e.g. `std::vector<float>`) and compiles against one version of ROOT; changing ROOT version breaks binary compatibility.

**uproot:** reimplementation of ROOT I/O in Python+Numpy with good performance, skipping a step that is unnecessary for filling arrays:

```
bytes of ROOT file → C++ objects → Numpy arrays
```

(disclosure: I’m the author)
Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):

$ pip install uproot --user
Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):

$ pip install uproot --user

One-per-event values become Numpy arrays

```python
>>> import uproot
>>> events = uproot.open("NanoAOD.root")["Events"]
>>> events.array("MET_pt")
array([17.244043, 42.480724, 29.10839 , ..., 12.228868, 44.115654, 33.511974], dtype=float32)
```

Multi-per-event become "jagged arrays," variable-length sublists simulated by indexes

```python
>>> events.array("Jet_pt")
jaggedarray([[[41.46875 27.625 22. 18.734375],
              [23.75 23.64062 18.9687 ... 16.7812 16.2812 16.25 ]],
              ...,
              [],
              [[34.03125 18.828125 18.359375]]])
```
Numpy arrays and jagged arrays

Installs without ROOT (or any particular version of ROOT):

$ pip install uproot --user

One-per-event values become Numpy arrays

```python
>>> import uproot
>>> events = uproot.open("NanoAOD.root")["Events"]
>>> events.array("MET_pt")
array([17.244043, 42.480724, 29.10839 , ..., 12.228868, 44.115654,
      33.511974], dtype=float32)
```

Multi-per-event become “jagged arrays,” variable-length sublists simulated by indexes

```python
>>> events.array("Jet_pt")
jaggedarray([[41.46875 27.625 22. 18.734375],
             [23.75 23.64062 18.9687 ... 16.7812 16.2812 16.25 ],
             ..., [[],
             [34.03125 18.828125 18.359375]])
```
Loading and computing columnar arrays is fast

<table>
<thead>
<tr>
<th>RAM memory</th>
<th>time to complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 MB</td>
<td>PyROOT load and compute</td>
</tr>
<tr>
<td>100 MB</td>
<td>JaggedArray compute in Python for loops</td>
</tr>
<tr>
<td>10 MB</td>
<td>root_numpy compute in loop over ufuncs</td>
</tr>
<tr>
<td>10 MB</td>
<td>root_numpy load</td>
</tr>
<tr>
<td>10 MB</td>
<td>serialized JSON text (for reference)</td>
</tr>
<tr>
<td>3 MB</td>
<td>JaggedArray of Table of pt, eta, phi</td>
</tr>
</tbody>
</table>

- Python list of lists of dicts
- root_numpy's array of arrays
- Python list of lists of `__slots__` classes
- `std::vector<std::vector<struct>>`
- loading and computing (jagged) \( pz = pt \cdot \sinh(\eta) \)
Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree)
  Prat�ush Das
  (DIANA-HEP fellow)

- Jagged array algorithms (vectorized on GPU)
  Jaydeep Nandi
  (Google Summer of Code)
Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree)  
  Pratyush Das  
  (DIANA-HEP fellow)

- Jagged array algorithms (vectorized on GPU)  
  Jaydeep Nandi  
  (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

\[
\text{metphi} - \quad \text{jetphi} \rightarrow \quad \text{phidiff}
\]

<table>
<thead>
<tr>
<th>flat</th>
<th>jagged</th>
<th>jagged</th>
</tr>
</thead>
</table>

17 / 24
Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree)  
  Pratyush Das  
  (DIANA-HEP fellow)

- Jagged array algorithms (vectorized on GPU)  
  Jaydeep Nandi  
  (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

```
metphi - jetphi → phidiff
       
flat    jagged   jagged
```

```
events[selection] → fewer_events
       
jagged  flat booleans  jagged
```
Work in progress by summer students

- Writing data with uproot (TH* and hopefully TTree)
  Pratyush Das
  (DIANA-HEP fellow)

- Jagged array algorithms (vectorized on GPU)
  Jaydeep Nandi
  (Google Summer of Code)

Logical extensions of Numpy for non-flat data:

\[
\text{metphi} - \text{jetphi} \rightarrow \text{phidiff}
\]

\[
\begin{align*}
\text{events}[\text{selection}] & \rightarrow \text{fewer events} \\
\text{jagged} & \rightarrow \text{flat booleans} \\
\text{jagged} & \rightarrow \text{jagged}
\end{align*}
\]

\[
\text{particle_attrs}.\text{max()} \rightarrow \text{event_attrs}
\]

\[
\begin{align*}
\text{jagged} & \rightarrow \text{reducer} \\
\text{flat} & \rightarrow \text{flat}
\end{align*}
\]
Nearly all histogramming packages designed for Numpy arrays are feature-poor (from a particle physicist’s perspective).

Access bin values, errors, weighted events, profile plots, efficiencies...
Nearly all histogramming packages designed for Numpy arrays are feature-poor (from a particle physicist’s perspective).

Access bin values, errors, weighted events, profile plots, efficiencies...
Another two Python+Numpy packages (again, I’m the author)

$ pip install histbook vegascope --user
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I’m the author)
$ pip install histbook vegascope --user

All histograms are N-dimensional and expressions to compute are next to their binning.

```python
>>> from histbook import *
>>> hist = Hist(
...     bin("sqrt(x**2 + y**2)", 5, 0, 1),
...     bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)"),
...     hist.step("arctan2(y, x)"))
```

Plots appear inline in JupyterLab or can be viewed
with VegaScope, a TCanvas-clone for Vega-Lite plots.
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I’m the author)
$ pip install histbook vegascope --user

All histograms are N-dimensional and expressions to compute are next to their binning.

```python
>>> from histbook import *
>>> hist = Hist(
...    bin("sqrt(x**2 + y**2)", 5, 0, 1),
...    bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)"),
...    hist.step("arctan2(y,x)"))
```

Projections, rebinning, binwise selections, can all be performed after filling.

```python
>>> (hist.select("arctan2(y, x) >= -pi")
...    .stack("arctan2(y, x)"")
...    .area("sqrt(x**2+y**2)"))
```
Histogramming in histbook, plotting in Vega-Lite

Another two Python+Numpy packages (again, I’m the author)
$ pip install histbook vegascope --user

All histograms are N-dimensional and expressions to compute are next to their binning.

```python
>>> from histbook import *
>>> hist = Hist(
...    bin("sqrt(x**2 + y**2)", 5, 0, 1),
...    bin("arctan2(y, x)", 3, -pi, pi))
>>> hist.fill(dataframe)
>>> beside(hist.step("sqrt(y**2 + x**2)")
...    hist.step("arctan2(y,x)"))

Projections, rebinning, binwise selections, can all be performed after filling.

```python
>>> (hist.select("arctan2(y, x) >= -pi")
...    .stack("arctan2(y, x)")
...    .area("sqrt(x **2+y**2)"))

Plots appear inline in JupyterLab or can be viewed with VegaScope, a TCanvas-clone for Vega-Lite plots.
Pandas is a bigger thing than Spark

Google Trends

"Pandas DataFrame"

"Spark DataFrame"

"ROOT TTree"

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$
Pandas is an in-memory table with advanced indexing.

Our in-memory, indexed data are histograms. Multidimensional binning is a multi-tiered index, which has the same structure whether dense or sparse.
Multi-tier indexing can also be used to represent jaggedness

```python
import uproot
events = uproot.open("NanoAOD.root")["Events"]
df = events.pandas.df(["MET_*", "Muon_*"])
df

<table>
<thead>
<tr>
<th>entry</th>
<th>subentry</th>
<th>MET_pt</th>
<th>MET_phi</th>
<th>Muon_pt</th>
<th>Muon_eta</th>
<th>Muon_phi</th>
<th>Muon_dxy</th>
<th>Muon_dz</th>
<th>Muon_charge</th>
<th>Muon_pdgId</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>17.244043</td>
<td>1.035400</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>42.480724</td>
<td>0.128357</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>29.108391</td>
<td>0.932129</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>29.060146</td>
<td>1.775635</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>39.994896</td>
<td>-1.418701</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>47.382843</td>
<td>2.964844</td>
<td>4.662842</td>
<td>0.839111</td>
<td>1.189697</td>
<td>-0.001610</td>
<td>0.994629</td>
<td>-1.0</td>
<td>13.0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>15.256557</td>
<td>1.333740</td>
<td>3.446648</td>
<td>-2.194824</td>
<td>-2.062500</td>
<td>-0.011620</td>
<td>-2.531250</td>
<td>-1.0</td>
<td>13.0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>17.382278</td>
<td>-2.448730</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>25.201349</td>
<td>-1.796387</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>24.990273</td>
<td>-1.426284</td>
<td>0.974159</td>
<td>-1.905760</td>
<td>1.094630</td>
<td>0.994310</td>
<td>0.995292</td>
<td>-1.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>
```
Multi-tier indexing can also be used to represent jaggedness

```python
df.pivot_table(index="entry", columns="subentry")
```

<table>
<thead>
<tr>
<th>subentry</th>
<th>MET_phi</th>
<th>MET_pt</th>
<th>Muon_charge</th>
<th>Muon_dxy</th>
<th>Muon_dz</th>
<th>Muon_eta</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MET_phi</td>
<td>MET_pt</td>
<td>Muon_charge</td>
<td>Muon_dxy</td>
<td>Muon_dz</td>
<td>Muon_eta</td>
</tr>
<tr>
<td>0</td>
<td>1.035400</td>
<td>17.244043</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>0.128357</td>
<td>42.480724</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>0.932129</td>
<td>29.108391</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>1.775635</td>
<td>29.060146</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>-1.418701</td>
<td>39.994896</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>2.964844</td>
<td>47.382843</td>
<td>-1.0</td>
<td>-0.001610</td>
<td>-0.011620</td>
<td>0.994629</td>
</tr>
<tr>
<td>6</td>
<td>1.333740</td>
<td>15.256557</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>-2.448730</td>
<td>17.382278</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>8</td>
<td>-1.796387</td>
<td>25.201349</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>9</td>
<td>1.400391</td>
<td>24.369278</td>
<td>-1.0</td>
<td>NaN</td>
<td>-0.004219</td>
<td>NaN</td>
</tr>
<tr>
<td>10</td>
<td>-0.028637</td>
<td>37.702976</td>
<td>NaN</td>
<td>NaN</td>
<td>-0.004219</td>
<td>NaN</td>
</tr>
<tr>
<td>11</td>
<td>-3.036133</td>
<td>17.890409</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>12</td>
<td>-1.971997</td>
<td>20.142917</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Scikit-HEP is a GitHub organization to build a Pythonic HEP ecosystem: scikit-hep.org

- uproot, histbook, vegascope
- root_numpy, root_pandas
- numpythia: bindings from Pythia to Numpy
- pyjet: bindings from FASTJET to Numpy
- formulate: convert between TTree::Draw syntax and numexpr (for histbook)
- decaylanguage: express complex $B$ decay chains

Onboarding or in development...

- hepvector: LorentzVector operations as Numpy ufuncs
- root_ufunc: use any ROOT function as a Numpy ufunc
The world beyond particle physics has a lot of useful software, but it’s not a perfect match to our needs.

To gain the benefits without giving up our own tools, we’re developing software to connect the two worlds.