



Thicket: Seeing the performance experiment forest for the individual run trees

RADIUSS Tutorial Series 2023

14 August 2023



Stephanie Brink, Olga Pearce



LLNL-PRES-850268

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC



Welcome to the RADIUSS AWS Tutorial Series!

Go to:

<https://software.llnl.gov/radiuss/event/2023/07/11/radiuss-on-aws/>

to learn more about our other tutorials and documentation!

Upcoming Tutorials

Date	Time (Pacific)	Project
August 3, 2023	9:00a.m.–11:00a.m.	 Build, link, and test large-scale applications with BLT
August 8–9 2023	8:00a.m.–11:30a.m. both days	 Learn to install your software quickly with Spack
August 10, 2023	9:00a.m.–11:00a.m.	 Use MFEM for scalable finite element discretization application development
August 14, 2023	9:00a.m.–12:00p.m.	 Integrate performance profiling capabilities into your applications with Caliper
		 Analyze hierarchical performance data with Hatchet
		 Optimize application performance on supercomputers with Thicket
August 17, 2023	9:00a.m.–11:00a.m.	 Use RAJA to run and port codes quickly across NVIDIA, AMD, and Intel GPUs
		 Discover, provision, and manage HPC memory with Umpire
August 22, 2023	9:00a.m.–11:00a.m.	 Visualize and analyze your simulations in situ with Ascent
August 24, 2023	9:00a.m.–11:00a.m.	 Leverage robust, flexible software components for scientific applications with Axiom
August 29, 2023	9:00a.m.–11:00a.m.	 Analyze runs of your code with WEAVE <small>Open Source Lawrence Livermore National Laboratory</small>
August 31, 2023	9:00a.m.–11:00a.m.	 Learn to run thousands of jobs in a workflow with Flux

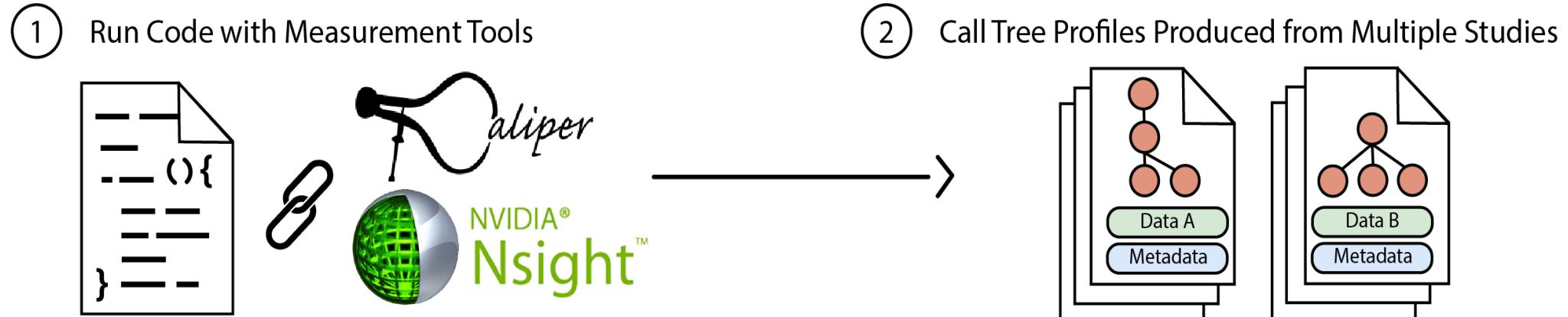


Tutorial Materials: <https://github.com/llnl/thicket-tutorial>

- The container includes example Jupyter notebooks, Thicket install 2023.2.0, and RAJA Performance Suite datasets in a self-contained Binder environment with all dependencies
 - Join our mailing list! <https://bit.ly/caliper-thicket-users>
- We'll use this material in the hands-on portion of the tutorial.

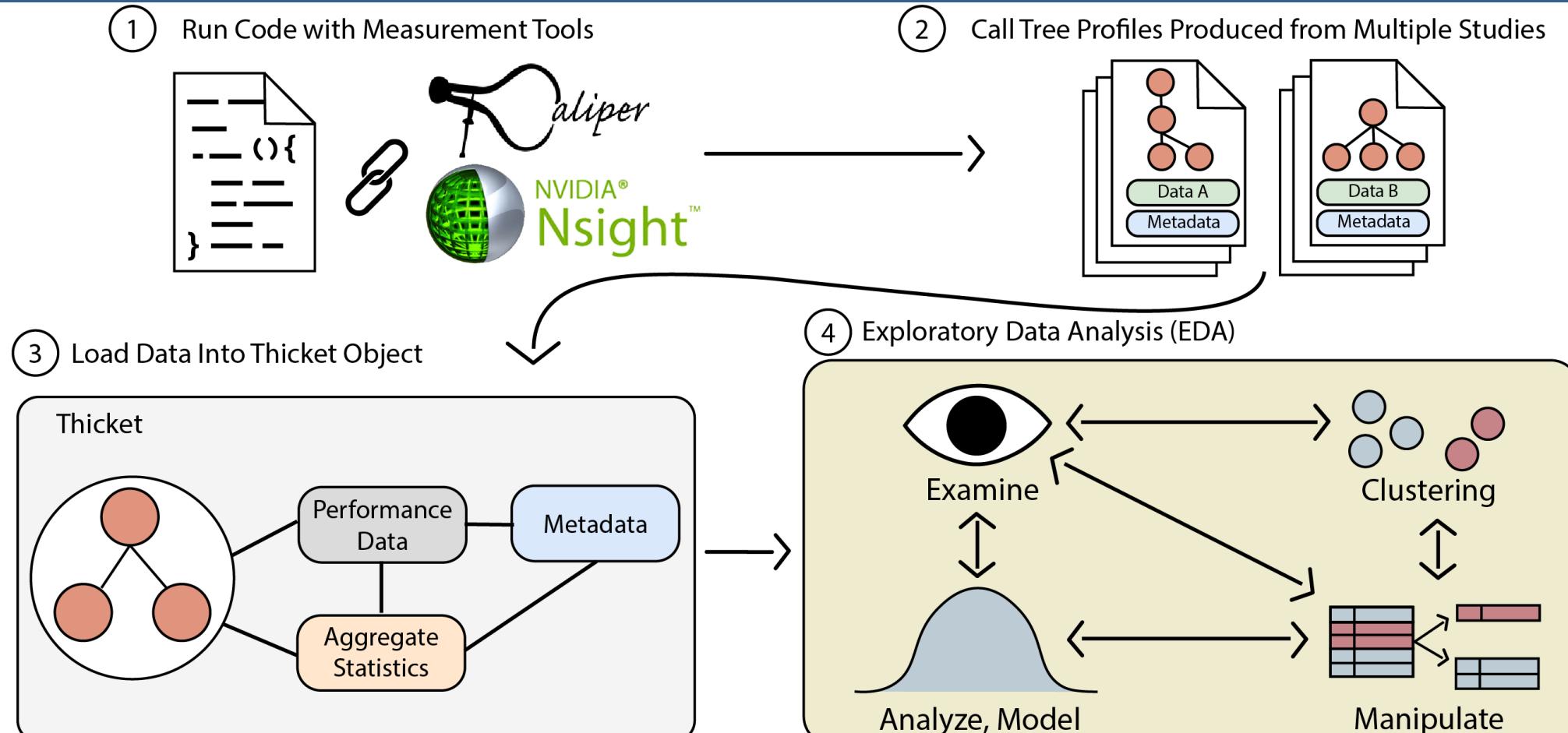
Challenge: Performance analysis in complex HPC ecosystem

- HPC software and hardware are increasingly complex. Need to understand:
 - Strong scaling and weak scaling of applications
 - Impact of application parameters on performance
 - Impact of choice of compilers and optimization levels
 - Performance on different hardware architectures (e.g., CPUs, GPUs)
 - Different tools to measure different aspects of application performance



Goal: Analyze and visualize performance data from different sources and types

Our big picture solution for analyzing and visualizing performance data from different sources and type



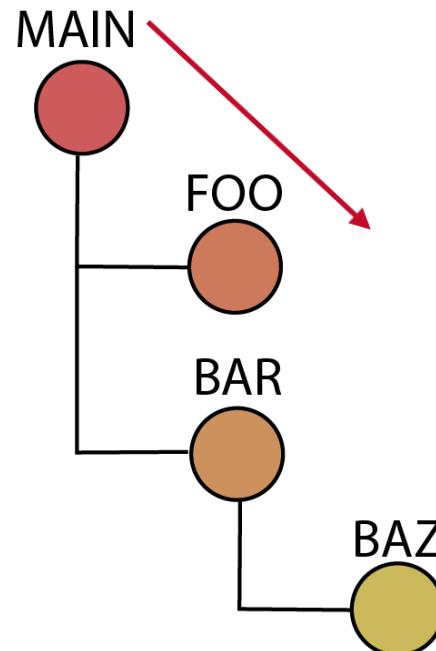
<https://github.com/LLNL/thicket>

<https://github.com/LLNL/thicket-tutorial>

What do profiling tools collect per run?



1) Call Tree



2) Performance data

Node	Cache Misses
MAIN	
FOO	
BAR	
BAZ	

- Time, FLOPS
- Cache misses
- Memory accesses

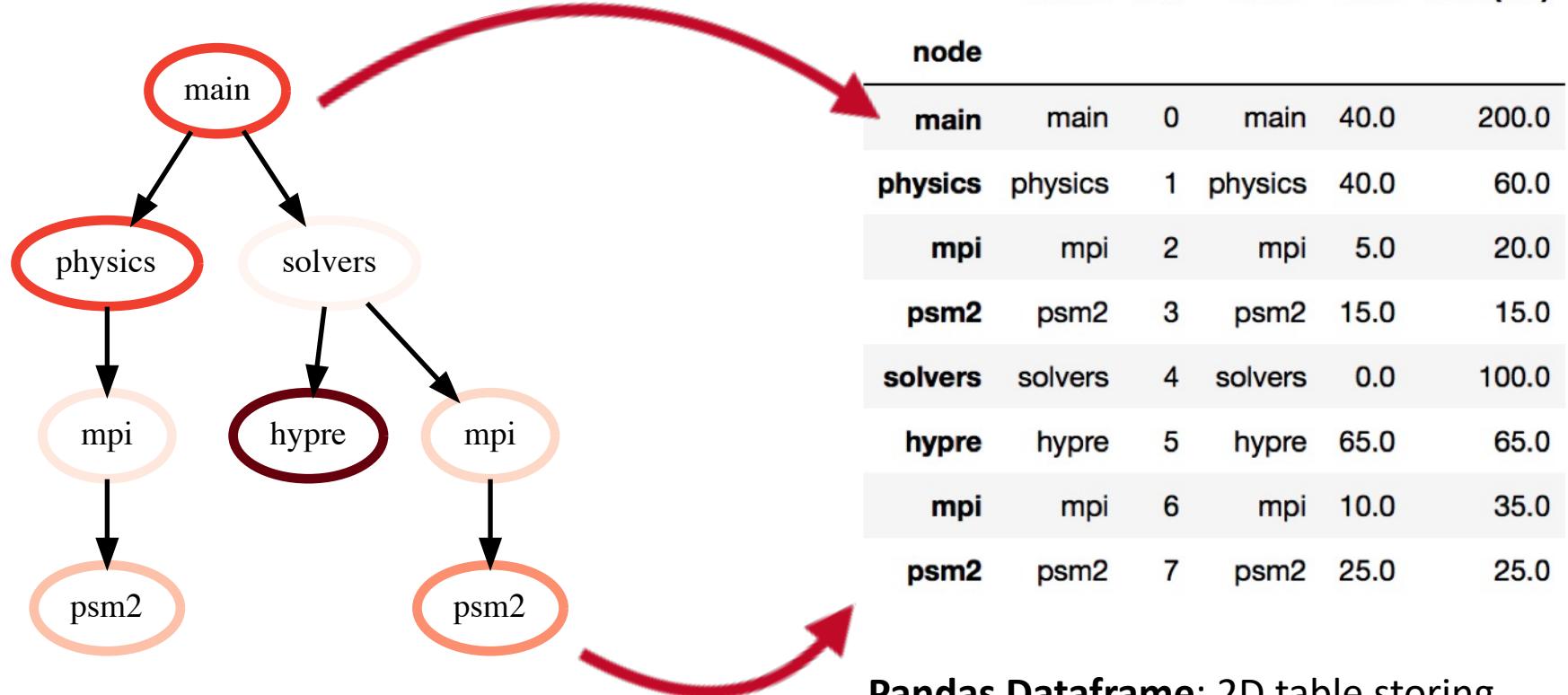
3) Metadata per run

User	Platform

- Batch submission (user, launch date)
- Hardware info (platform)
- Build info (compiler versions/flags)
- Runtime info (problem parameters, number of MPI ranks used)



Thicket builds upon Hatchet's *GraphFrame*: a Graph and a Dataframe



Graph: Stores relationships between parents and children

Pandas Dataframe: 2D table storing numerical data associated with each node (may be unique per rank, per thread)



<https://github.com/LLNL/hatchet>
<https://github.com/LLNL/hatchet-tutorial>



Visualizing Hatchet's GraphFrame components

```
>>> print(gf.tree())      # print graph  
>>> print(gf.dataframe)  # print dataframe
```

```
0.000 foo  
|   └ 6.000 bar  
|       |   └ 5.000 baz  
|   └ 0.000 qux  
|       |   └ 5.000 quux  
|           |   └ 10.000 corge  
|           |   └ 15.000 garply  
|               |   └ 1.000 grault  
15.000 waldo  
|   └ 3.000 fred  
|       |   └ 5.000 plugh  
|   └ 15.000 garply
```

Legend (Metric: time)
■ 13.50 - 15.00
■ 10.50 - 13.50
■ 7.50 - 10.50
■ 4.50 - 7.50
■ 1.50 - 4.50
■ 0.00 - 1.50

name User code

◀ Only in left graph

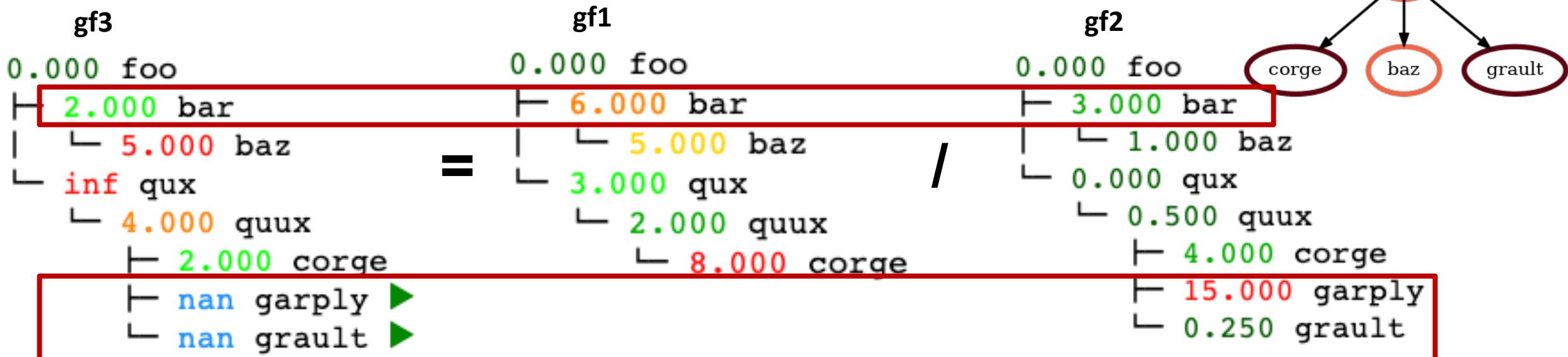
node	name	time	time (inc)
{'name': 'foo'}	foo	0.0	130.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'quux'}	quux	0.0	60.0
{'name': 'quux'}	quux	5.0	60.0
{'name': 'corge'}	corge	10.0	55.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'garply'}	garply	15.0	15.0
{'name': 'grault'}	grault	10.0	10.0

▶ Only in right graph

Compare GraphFrames using division (or add, subtract, multiply)

```
>>> gf3 = gf1 / gf2 # divide graphframes
```

*First, unify two trees since
structure is different



```
>>> gf3 = gf1 + gf2 # add graphframes  
>>> gf3 = gf1 - gf2 # subtract graphframes  
>>> gf3 = gf1 * gf2 # multiply graphframes
```

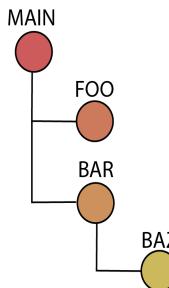


Use Thicket to *compose* performance profiles in Python

P1

Metadata

User	Platform



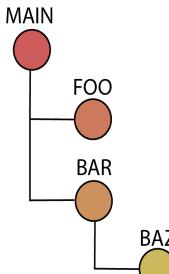
Performance metrics

Node	Cache Misses
MAIN	24
FOO	
BAR	
BAZ	

P2

Metadata

User	Platform

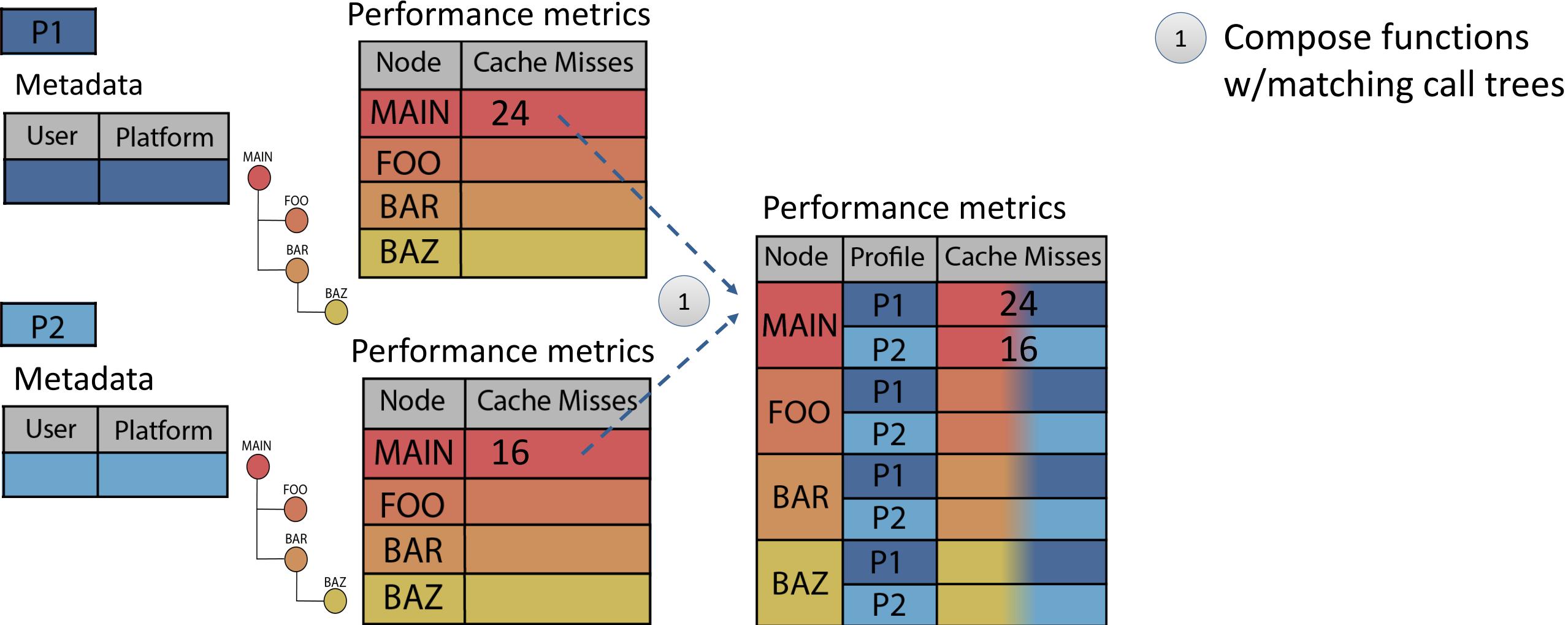


Performance metrics

Node	Cache Misses
MAIN	16
FOO	
BAR	
BAZ	

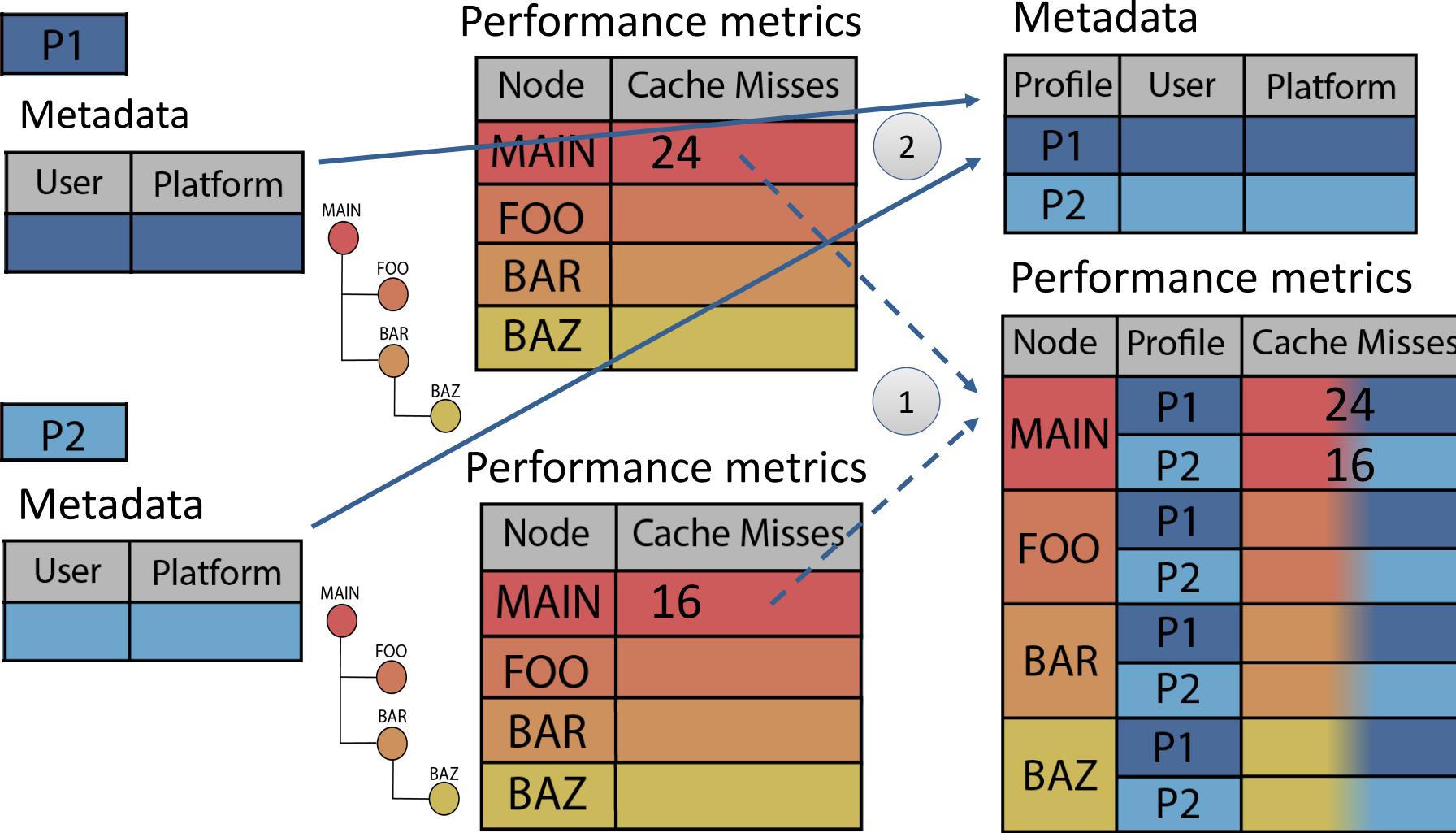


Use Thicket to *compose* performance profiles in Python





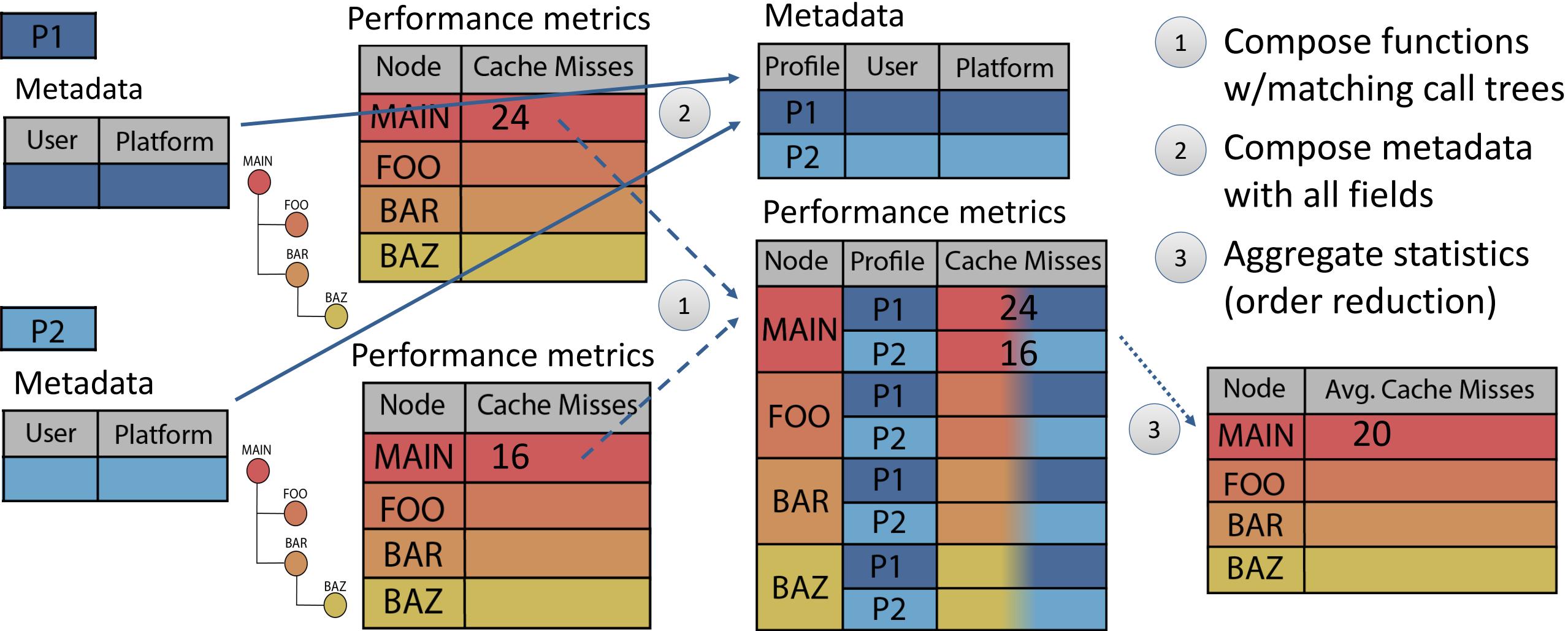
Use Thicket to *compose* performance profiles in Python



- 1 Compose functions w/matching call trees
- 2 Compose metadata with all fields



Use Thicket to *compose* performance profiles in Python





Thicket components are *interconnected*

Metadata

Profile	User	Platform
P1	Jon	lassen
P2	Bob	lassen

Performance metrics

Node	Profile	Cache Misses
MAIN	P1	red
	P2	blue
FOO	P1	orange
	P2	blue
BAR	P1	yellow
	P2	blue
BAZ	P1	green
	P2	blue

Filter on metadata:
platform=="lassen" &&
user=="Bob"

Filtered Metadata

Profile	User	Platform
P2	Bob	lassen

Filtered Performance metrics

Node	Profile	Cache Misses
MAIN	P2	red
FOO	P2	orange
BAR	P2	yellow
BAZ	P2	green

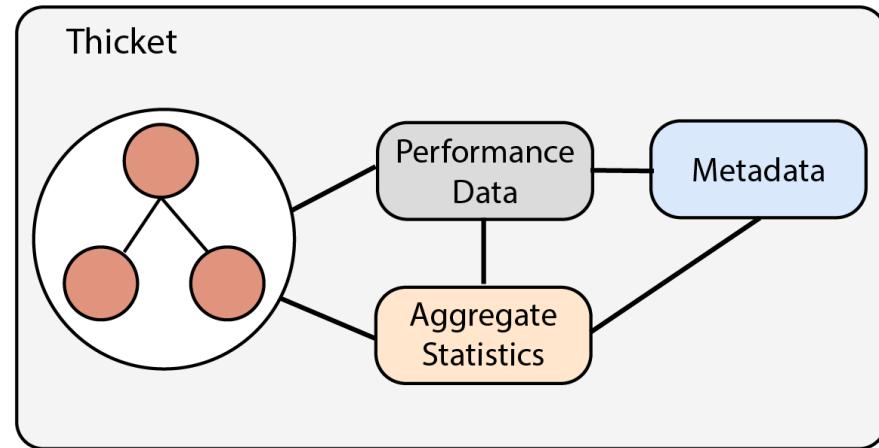
Metadata fields useful for understanding
and manipulating thicket object!



Thicket enables exploratory data analysis of multi-run data



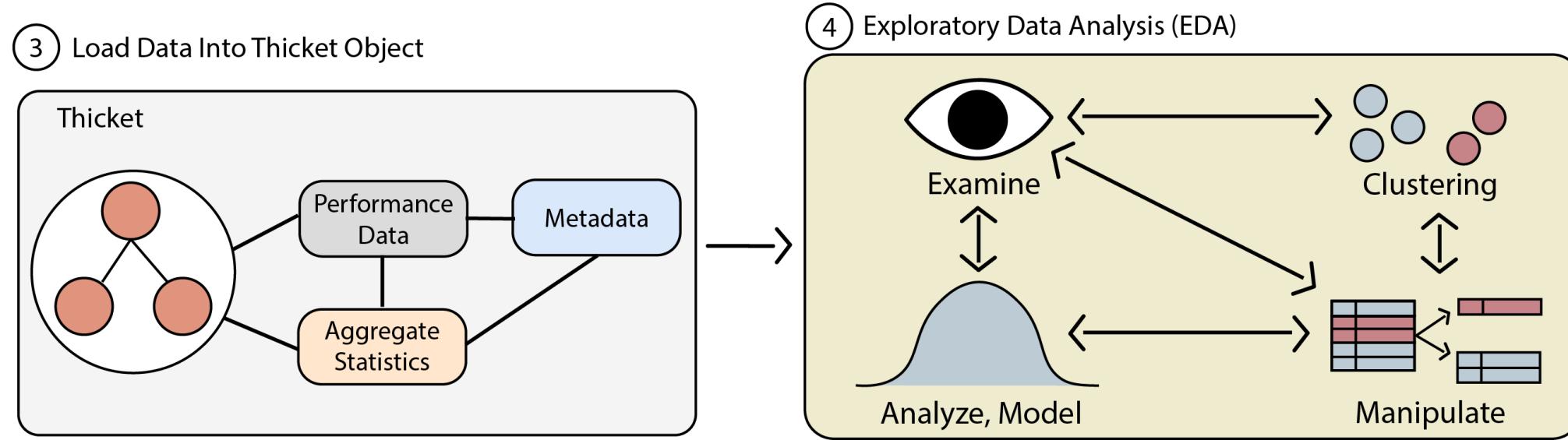
3 Load Data Into Thicket Object



- Compose data from diff. sources and types
 - Different scaling (e.g., strong, weak)
 - Different application parameters
 - Different compilers and optimization levels
 - Different hardware types (e.g., CPUs, GPUs)
 - Different performance tools



Thicket enables exploratory data analysis of multi-run data

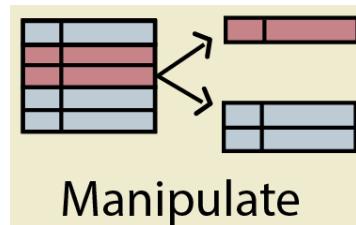


- Compose data from diff. sources and types
 - Different scaling (e.g., strong, weak)
 - Different application parameters
 - Different compilers and optimization levels
 - Different hardware types (e.g., CPUs, GPUs)
 - Different performance tools
- Perform analysis on the thicket of runs
 - Manipulate the set of data
 - Visualize the dataset
 - Perform analysis on the data
 - Model data
 - Leverage third-party tools in the Python ecosystem



- Open-source suite of loop-based kernels commonly found in HPC applications showcasing performance of different programming models on different hardware
- 560 runs/profiles:
 - 2 clusters (CPU, CPU+GPU)
 - 3 programming models (sequential, OpenMP, CUDA)
 - 4 problem sizes
 - 3 performance tools (Caliper, PAPI, Nsight Compute)
 - 3 compilers, 4 optimizations

cluster	systype build	problem size	compiler	compiler optimizations	omp num threads	cuda compiler	block sizes	RAJA variant	#profiles
0	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential 160
1	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential 160
2	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	-O0	72	N/A	N/A	OpenMP 40
3	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	-O0	72	N/A	N/A	OpenMP 40
4	lassen	blueos_3_ppc64le_ib_p9	[1M, 2M, 4M, 8M]	xlc++_r-16.1.1.12	-O0	1 nvcc-11.2.152	[128, 256, 512, 1024]	CUDA	160



Use Thicket to *compose* multi-platform, multi-tool data

Thicket object composed of 2 profiles run on CPU

		node	problem_size	time (exc)	Reps	Retiring	Backend bound
Apps_NODAL_ACCUMULATION_3D	1M	0.204583	100	0.144928	0.783786		
		0.795511	100	0.139002	0.788017		
	1M	0.067061	100	0.402238	0.510525		
	4M	0.241508	100	0.400775	0.515976		

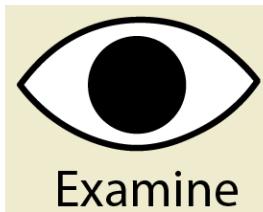
Thicket object composed of 2 profiles run on GPU

		node	problem_size	time (gpu)	gpu_compute_memory_throughput	gpu_dram_throughput	sm_throughput
Apps_NODAL_ACCUMULATION_3D	1M	0.007478			70.689752	46.724767	7.330745
		0.026951			74.275834	51.257993	7.688628
	1M	0.006028			81.012826	67.751194	35.676942
	4M	0.021422			91.929933	70.122011	35.386470
Apps_VOL3D							

CPU

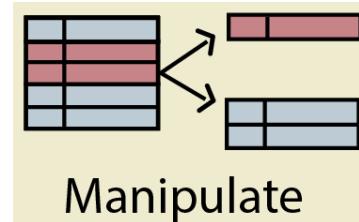
GPU

		node	problem_size	time (exc)	Reps	Retiring	Backend bound	time (gpu)	gpu_compute_memory_throughput	gpu_dram_throughput	sm__throughput
Apps_NODAL_ACCUMULATION_3D	1M	1M	76	0.204583	100	0.144928	0.783786	0.007478		70.689752	46.724767
		4M		0.795511	100	0.139002	0.788017	0.026951		74.275834	51.257993
	1M			0.067061	100	0.402238	0.510525	0.006028		81.012826	67.751194
	4M			0.241508	100	0.400775	0.515976	0.021422		91.929933	70.122011
Apps_VOL3D											



- Dataset: 4 types of profiles side-by-side to compare CPU to GPU performance
 - 1 Basic CPU metrics from Caliper
 - 2 Top-down metrics from Caliper/PAPI
 - 3 GPU runtime from Caliper
 - 4 GPU metrics from Nsight Compute
- Examples of analysis:
 - Compute CPU/GPU speedup
 - Correlate memory and compute usage on the CPU vs. GPU

Node	Problem size	CPU			CPU top-down		GPU		GPU Nsight Compute				Derived speedup
		time (exc)	Bytes/Rep	Flops/Rep	Retiring	Backend bound	time (gpu)	gpu_compute_memory_throughput	gpu_dram_throughput	sm_throughput	sm_warps_active		
Apps_VOL3D	8M	0.498815	282109496	632421288	0.377843	0.540604	0.040761	93.742058	72.140428	36.206767	54.459589	12.237556	
Lcals_HYDRO_1D	8M	2.077556	201326600	41943040	0.032965	0.909545	0.242928	92.944968	92.944968	6.595714	95.266148	8.552147	



Manipulate: Filter using call path query

```

0.001 Base_CUDA
└ 0.000 Algorithm
   └ 0.000 Algorithm_MEMORY
      └ 0.002 Algorithm_MEMORY.block_128
      └ 0.009 Algorithm_MEMORY.block_256
      └ 0.006 Algorithm_MEMORY.library
   └ 0.000 Algorithm_MEMSET
      └ 0.001 Algorithm_MEMSET.block_128
      └ 0.004 Algorithm_MEMSET.block_256
      └ 0.003 Algorithm_MEMSET.library
   └ 0.000 Algorithm_REDUCE_SUM
      └ 0.003 Algorithm_REDUCE_SUM.block_128
      └ 0.004 Algorithm_REDUCE_SUM.block_256
      └ 0.002 Algorithm_REDUCE_SUM.cub
   └ 0.000 Algorithm_SCAN
      └ 0.006 Algorithm_SCAN.default
  
```

Input call tree

Filter on call path:
(1) Node named
“Base_CUDA”

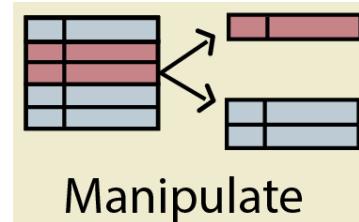
```

0.001 Base_CUDA
  
```

Output call tree

I Lumsden et al. “Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications”, e-Science 2022





Manipulate: Filter using call path query

```
0.001 Base_CUDA
└ 0.000 Algorithm
  └ 0.000 Algorithm_MEMORY
    └ 0.002 Algorithm_MEMORY.block_128
    └ 0.009 Algorithm_MEMORY.block_256
    └ 0.006 Algorithm_MEMORY.library
  └ 0.000 Algorithm_MEMSET
    └ 0.001 Algorithm_MEMSET.block_128
    └ 0.004 Algorithm_MEMSET.block_256
    └ 0.003 Algorithm_MEMSET.library
  └ 0.000 Algorithm_REDUCE_SUM
    └ 0.003 Algorithm_REDUCE_SUM.block_128
    └ 0.004 Algorithm_REDUCE_SUM.block_256
    └ 0.002 Algorithm_REDUCE_SUM.cub
  └ 0.000 Algorithm_SCAN
    └ 0.006 Algorithm_SCAN.default
```

Input call tree

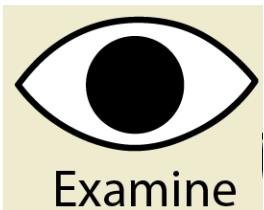
Filter on call path:

- (1) Node named “Base_CUDA”
- (2) Node with “block_128” in name (and any nodes in between)

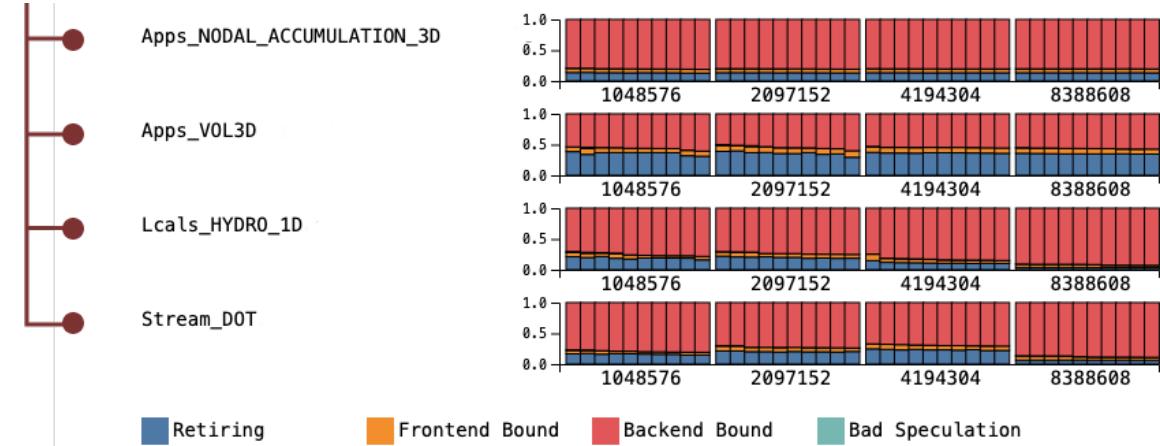
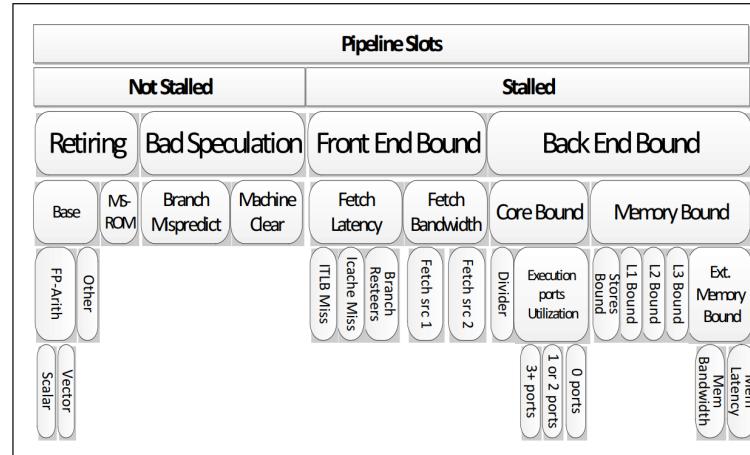
```
0.001 Base_CUDA
└ 0.000 Algorithm
  └ 0.000 Algorithm_MEMORY
    └ 0.002 Algorithm_MEMORY.block_128
  └ 0.000 Algorithm_MEMSET
    └ 0.001 Algorithm_MEMSET.block_128
  └ 0.000 Algorithm_REDUCE_SUM
    └ 0.003 Algorithm_REDUCE_SUM.block_128
```

Output call tree

I Lumsden et al. “Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications”, e-Science 2022



Visualize: Intel CPU top-down analysis

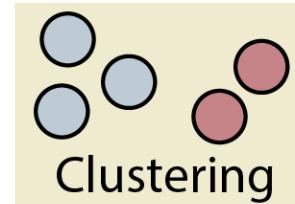


- *Top-down analysis* uses HW counters in a hierarchy to identify bottlenecks*
- Use Caliper's top-down module to derive top-down metrics for call-tree regions

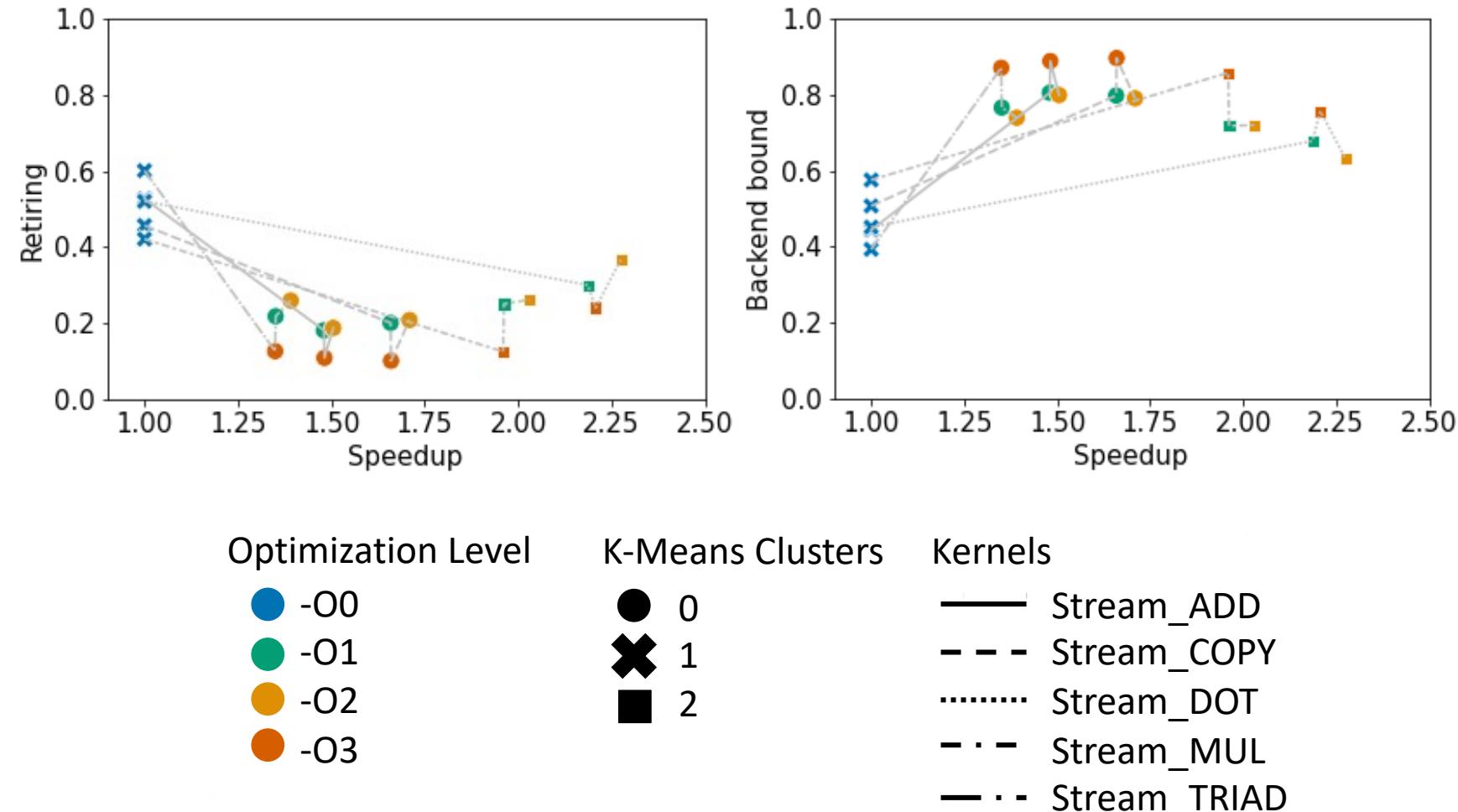
- Thicket's *tree+table* visualization shows top-down metrics as stacked bar charts, each bar is a profile
 - Apps_VOL3D has the highest retiring rates
 - Lcals_HYDRO and Stream_DOT become more backend bound as problem size grows

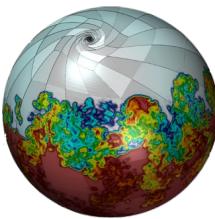
* Yasin, A.: A Top-Down Method for Performance Analysis and Counters Architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 35–44. IEEE, CA, USA (Mar 2014).

Use third-party Python libraries, e.g., Scikit-learn clustering



1. Select data of interest
 - Filter 8M problem size
 - Use query language to extract all implementations of the Stream kernel
2. (optional) Normalize data
3. Apply scikit-learn clustering to top-down analysis metrics of runs with different compiler optimization levels



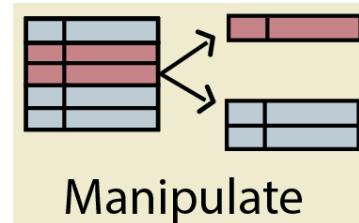


Case Study 2: MARBL multi-physics code

- MARBL is a next-generation multi-physics code developed at LLNL
- 60 runs/profiles:
 - 2 clusters (rztopaz, AWS ParallelCluster)
 - 2 MPI libraries (impi, openmpi)
 - 6 node/rank counts
 - 5 repeat runs per config

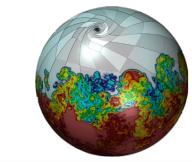
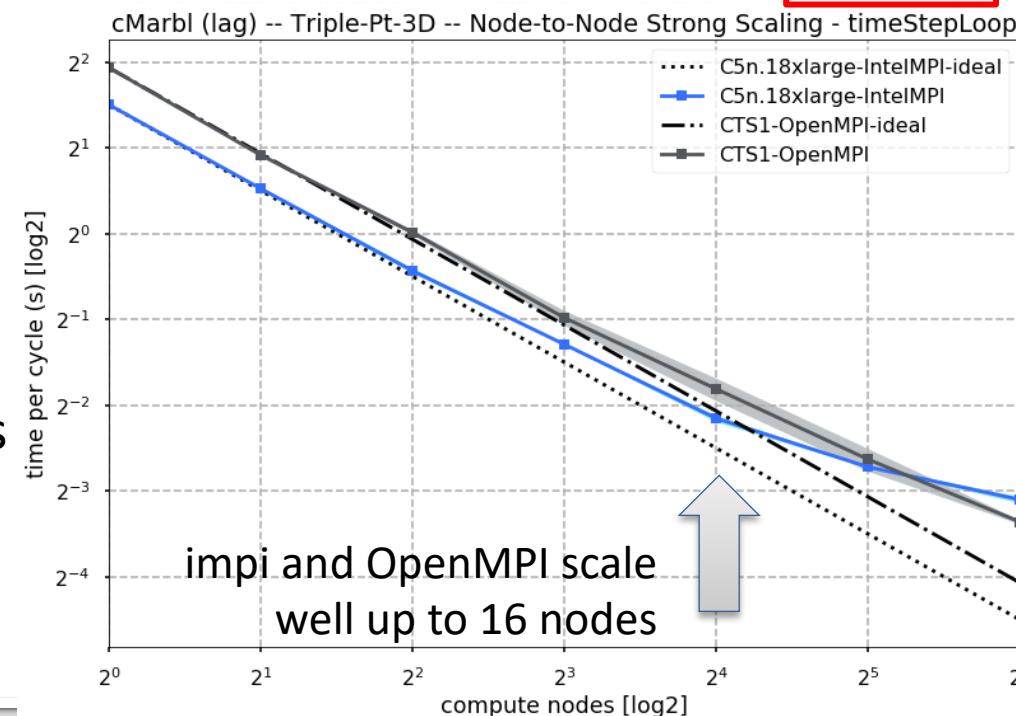
	cluster	ccompiler	mpi	version	numhosts	mpi.world.size	#profiles
0	ip----	/usr/tce/packages/clang/clang-9.0.0	impi	v1.1.0-203-gcb0efb3	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30
1	rztopaz	/usr/tce/packages/clang/clang-9.0.0	openmpi	v1.1.0-201-g891eaf1	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30

Manipulate: Compute noise and scaling



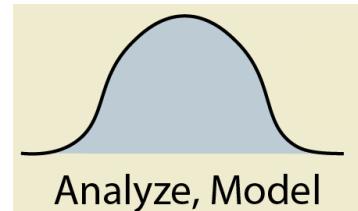
node	profile	Total time	name	mpi.world.size
{'name': 'main', 'type': 'function'}	-8554409769265002864	58036.664552	main	144
	-7335101512240609798	55318.808836	main	36
	-6029692086108825020	156984.246813	main	2304
	-5606382734792961361	64122.371533	main	288
	-4058809097109060732	155040.998627	main	2304
	-3193575964635936033	71010.504038	main	576
	-2978339073585311581	55910.708449	main	72
	-2939704488254773514	157934.204076	main	2304
	-2771797711381234985	56893.512948	main	144
	-2638513839856695106	97432.260966	main	1152

node	profile	Total time	name	mpi.world.size	
		-7335101512240609798	55318.808836	main	36
		-843517585394879415	55110.656885	main	36
	{'name': 'main', 'type': 'function'}	7720382918482619866	55155.581578	main	36
		8293335926964337960	55139.134916	main	36
		8335957980556391465	55013.682102	main	36



1. Use groupby(MPI.world.size) to generate unique subsets of data which are repeated runs; compute noise
2. Compose runs on different platforms and at different scales
3. Generate strong scaling plot with matplotlib
 - Deviation shown in shaded region, dots are average of 5 runs

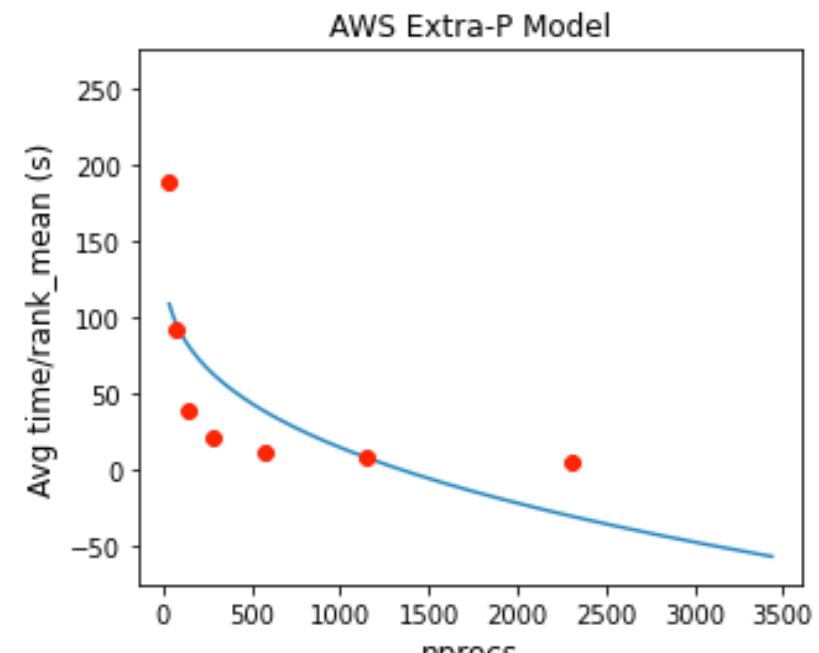
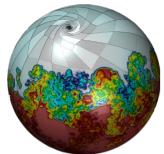
Model: Use third-party Python library, Extra-P



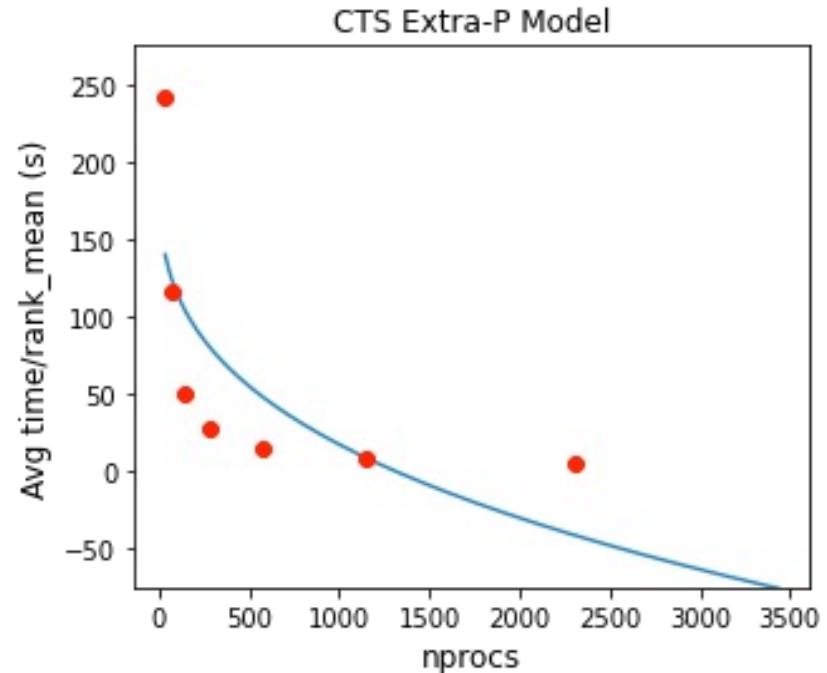
Extra-P derives an analytical performance model from an ensemble of profiles covering one or more modeling parameters

<http://github.com/extra-p/extrap>

- Select functions of interest
- Call Extra-P to model scaling on different hardware types



154.8848323145599 + -14.012557071778664 * p^(1/3)
● M_solver->Mult



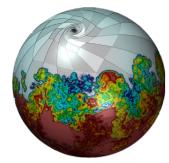
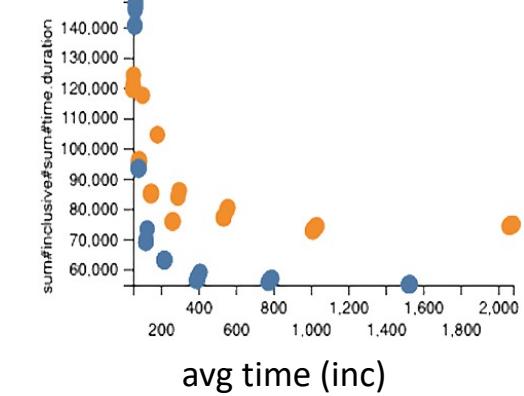
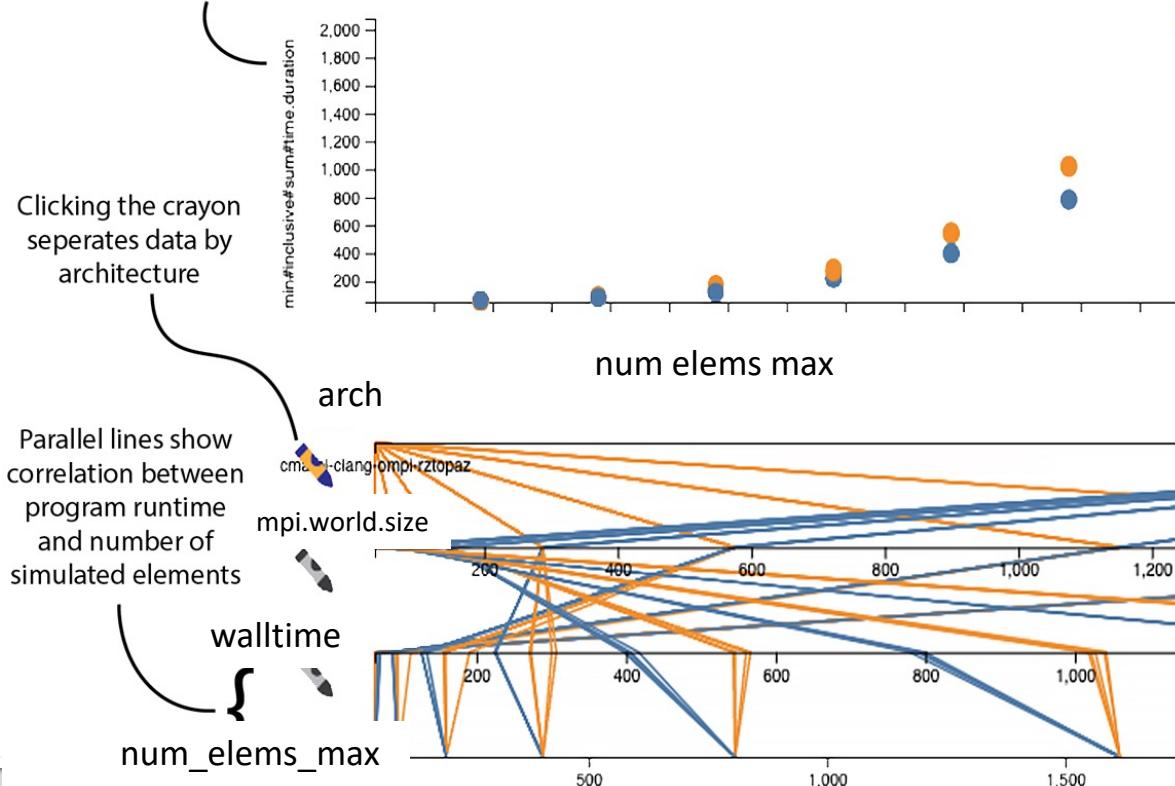
200.23124269331294 + -18.278533682209932 * p^(1/3)
● M_solver->Mult



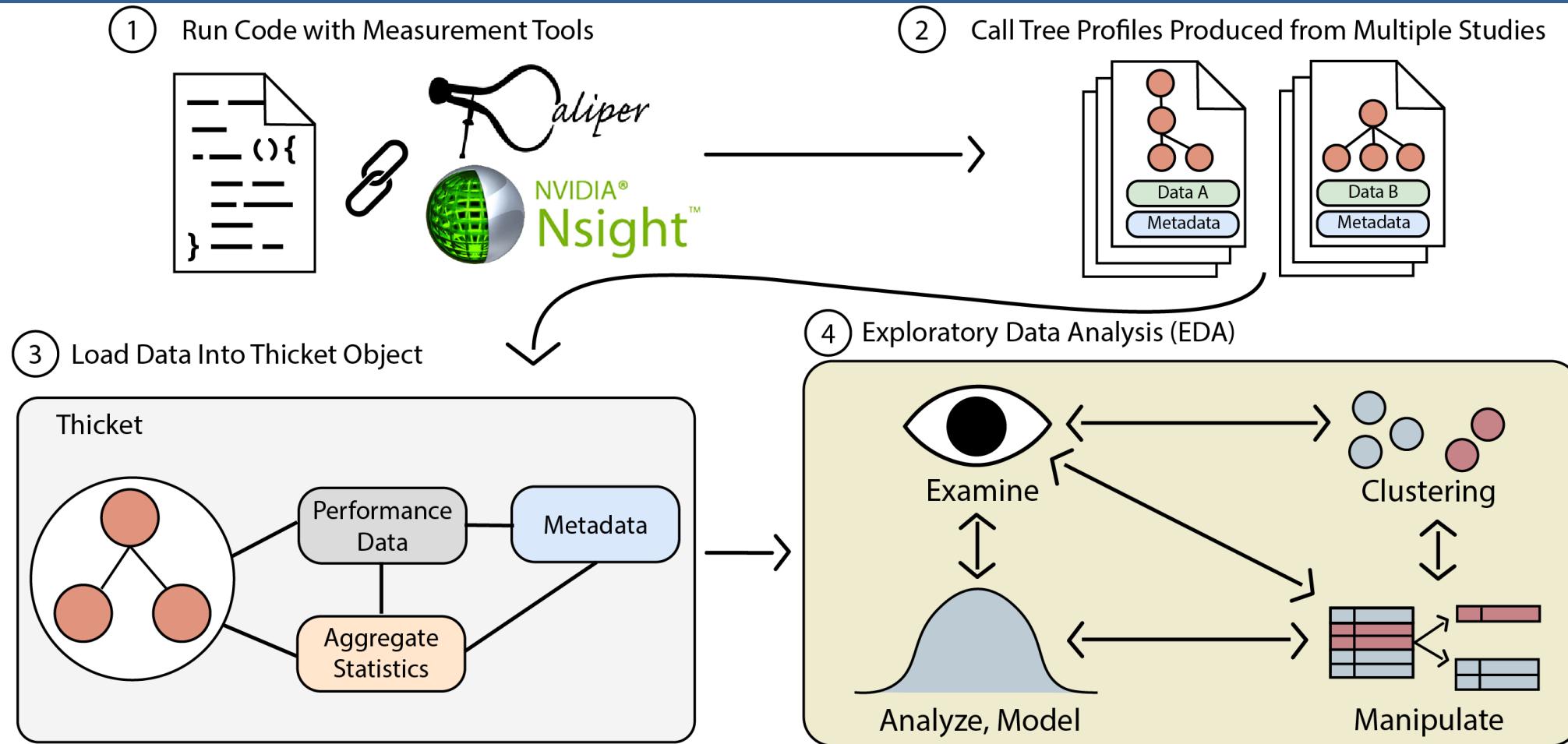
Visualize metadata with parallel coordinates plot

- Thicket's interactive parallel coordinates plot shows relationships between metadata variables, and between metadata and performance data

The metric values are associated with one node in the call tree.



Thicket is a toolkit for exploratory data analysis of multi-run data



<https://github.com/LLNL/thicket>

<https://github.com/LLNL/thicket-tutorial>

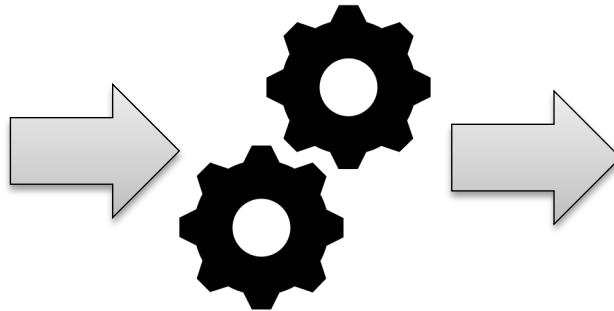
On LC Automated Application Performance Analysis Workflow

Go to <https://lc.llnl.gov/jupyter>

```
#include <caliper/cali.h>

static inline
void LagrangeElements(Domain&
domain, Index_t numElem)
{
    CALI_CXX_MARK_FUNCTION;
// ...
```

Caliper instrumentation
in the application



At runtime: Performance
and Metadata Collection

A screenshot of a Jupyter notebook interface. The title bar says 'jupyterhub hatchet-small-graph-example-casc-jan2021 Last Checkpoint: 2 hours ago (autosaved)'. The menu includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help. The toolbar has icons for file operations, Run, Stop, etc. A code cell shows Python code for reading Caliper files and creating dataframes, followed by a plot command. To the right of the notebook is a green logo featuring stylized leaves or petals.

Visualization and analysis of caliper datasets
using Python in Jupyter notebooks



c/o D Boehme

Jupyter and Thicket

Hands-On Time!

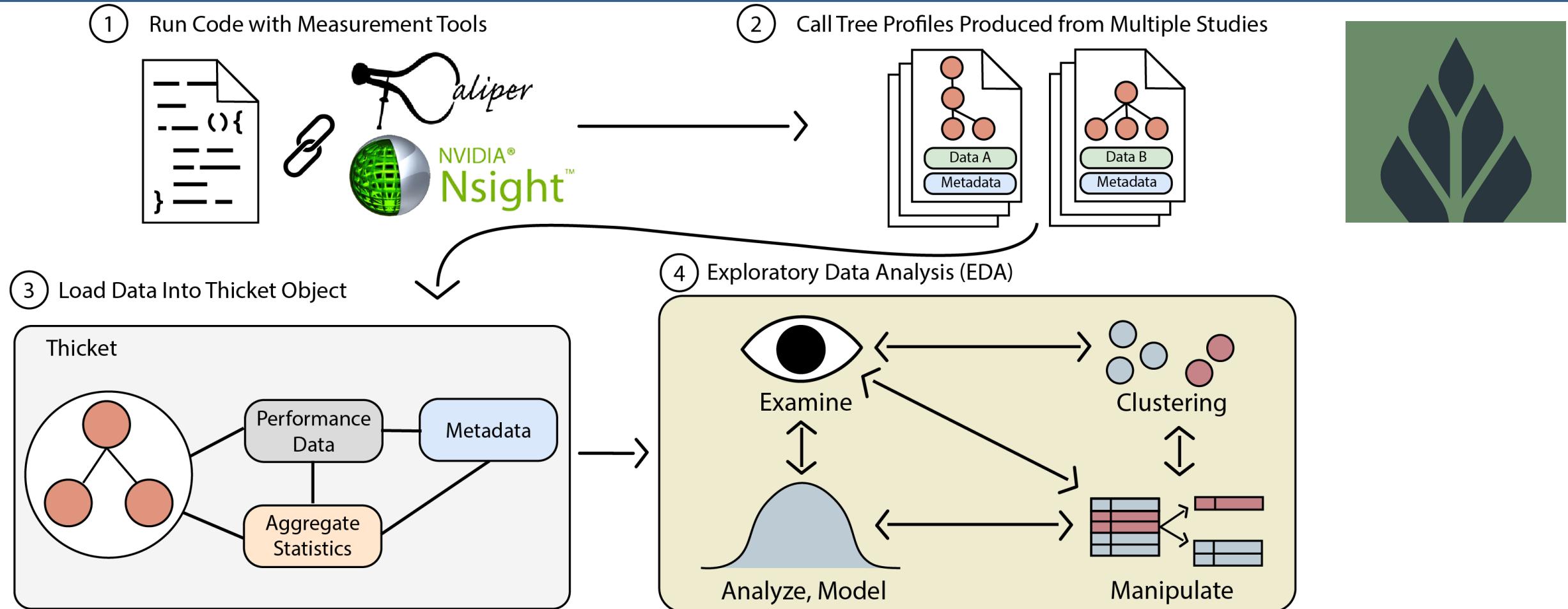
- The container includes example Jupyter notebooks, Thicket 2023.2.0 install, and RAJA Performance Suite datasets in a self-contained Binder environment with all dependencies
 - We expect to update our RAJA Performance Suite tutorial datasets by the end of September

Steps:

- Go to <https://github.com/llnl/thicket-tutorial>
- In the README, click the badge called “launch binder”
- We’ll start with notebooks/01_thicket_tutorial.ipynb

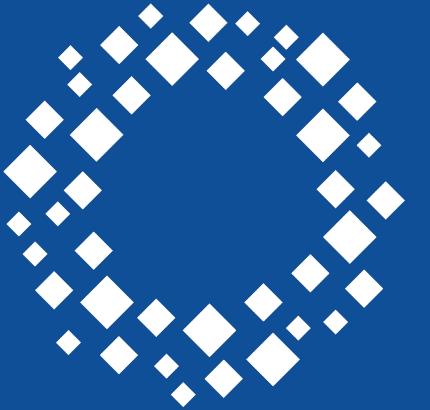
Join our mailing list! <https://bit.ly/caliper-thicket-users>

Thicket is a toolkit for exploratory data analysis of multi-run data



Join our mailing list! <https://bit.ly/caliper-thicket-users>





CASC

Center for Applied
Scientific Computing



**Lawrence Livermore
National Laboratory**

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.