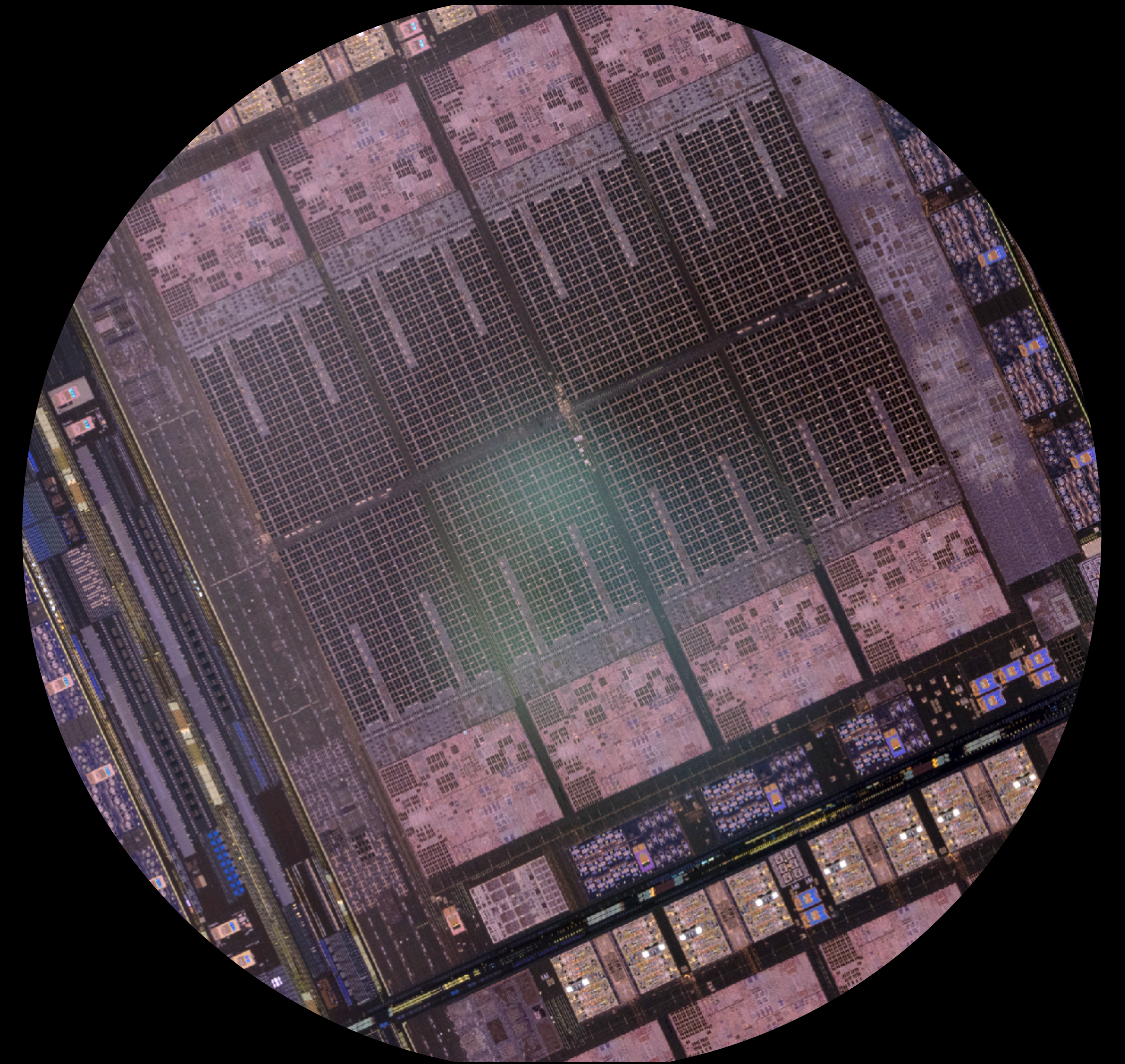
Data-Driven Design:

Leveraging a custom CPython data model for high-performance microprocessor design.



R. Taggart, N. Hieter, K. Kalafala
IBM Electronic Design Automation (EDA)

30 April 2022

I.

How to build a microprocessor: a **Big Data Problem**

II.

An efficient **CPython** data model

III.

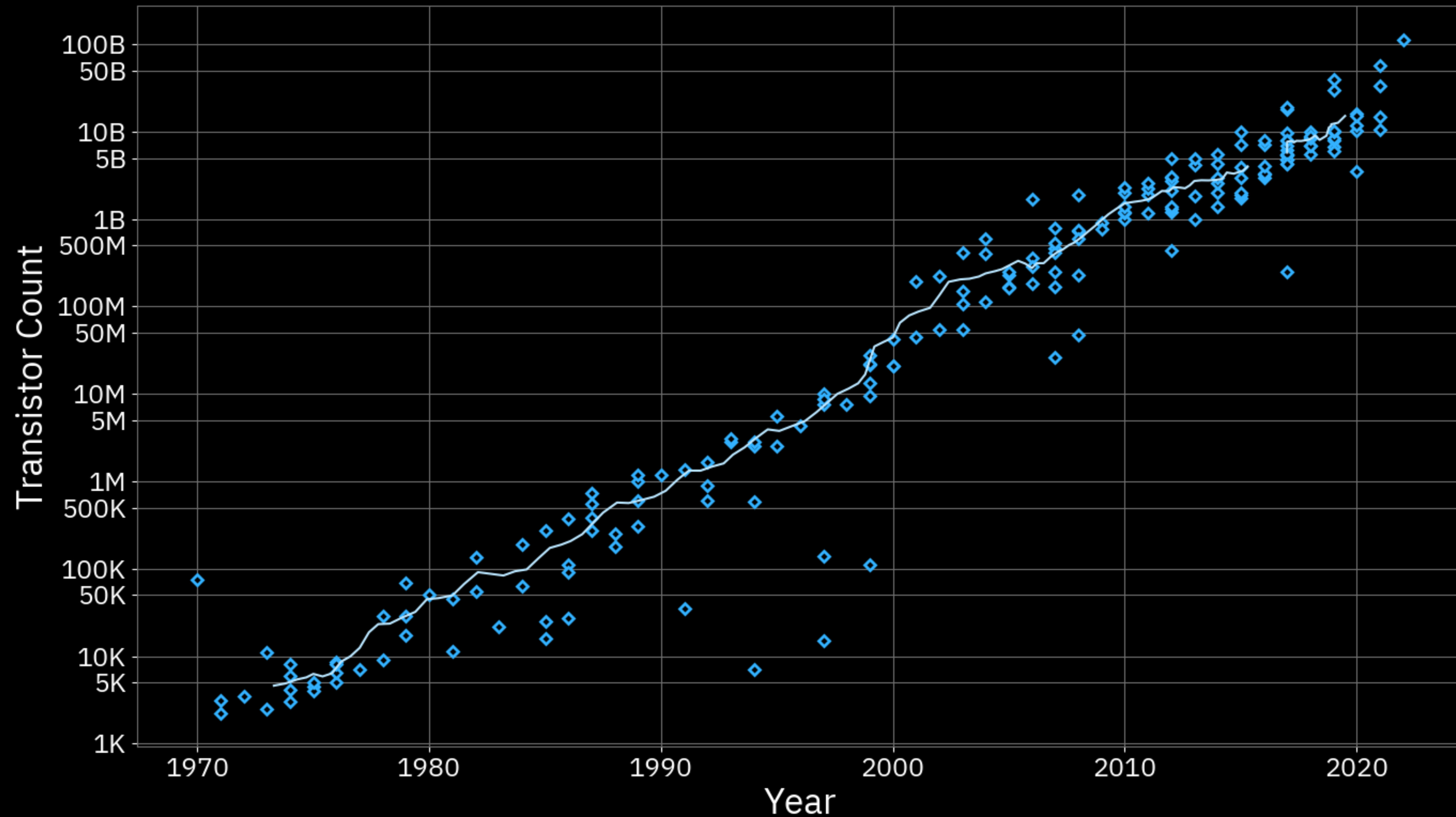
Using **Python** for data analysis

IV.

Lessons learned & Wrap-up

History of Transistor Count in Microprocessors

Moore's law: transistor count doubles every two years

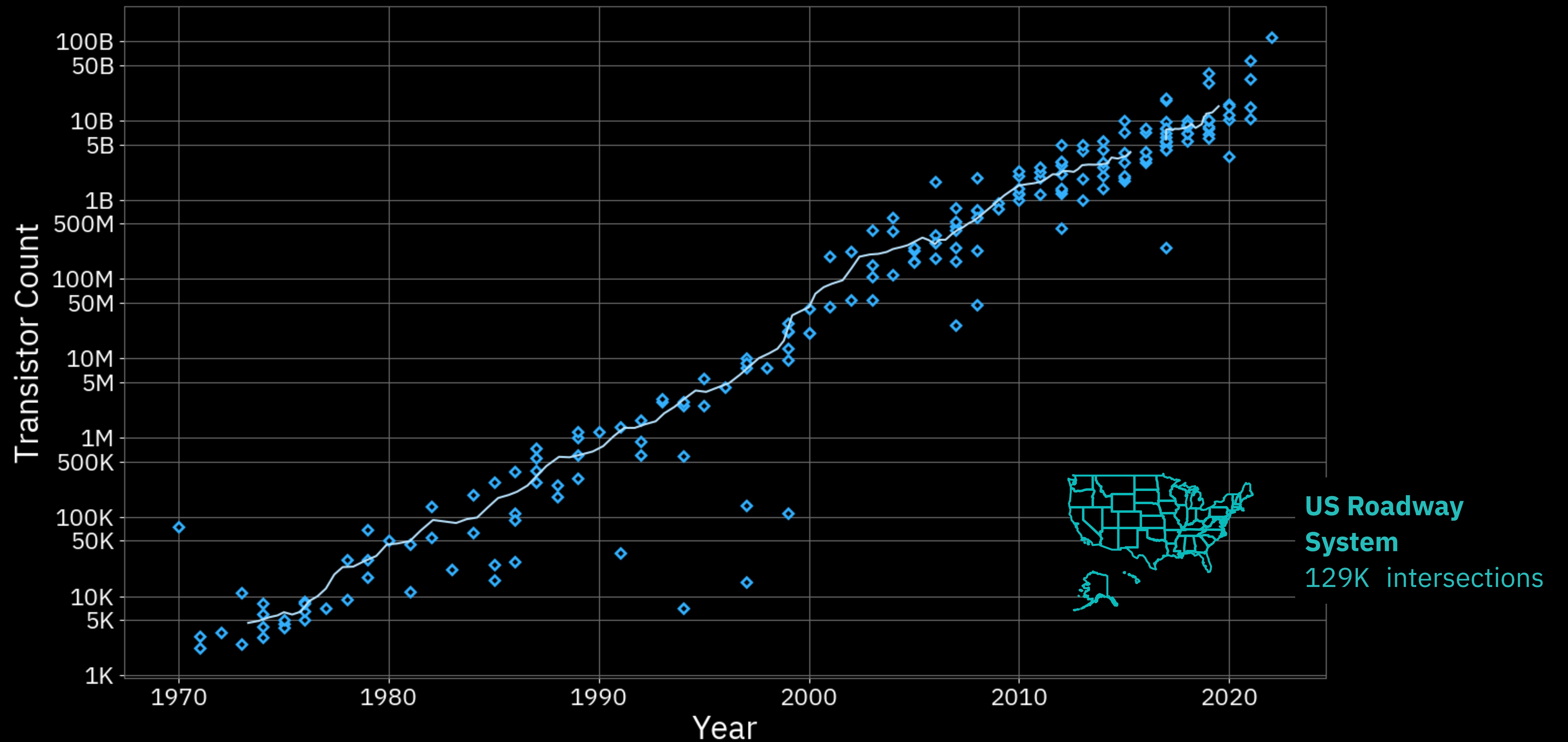


Design automation tools are required to build microprocessors!

Source: wikipedia.org/en/transistor_count

History of Transistor Count in Microprocessors

Moore's law: transistor count doubles every two years

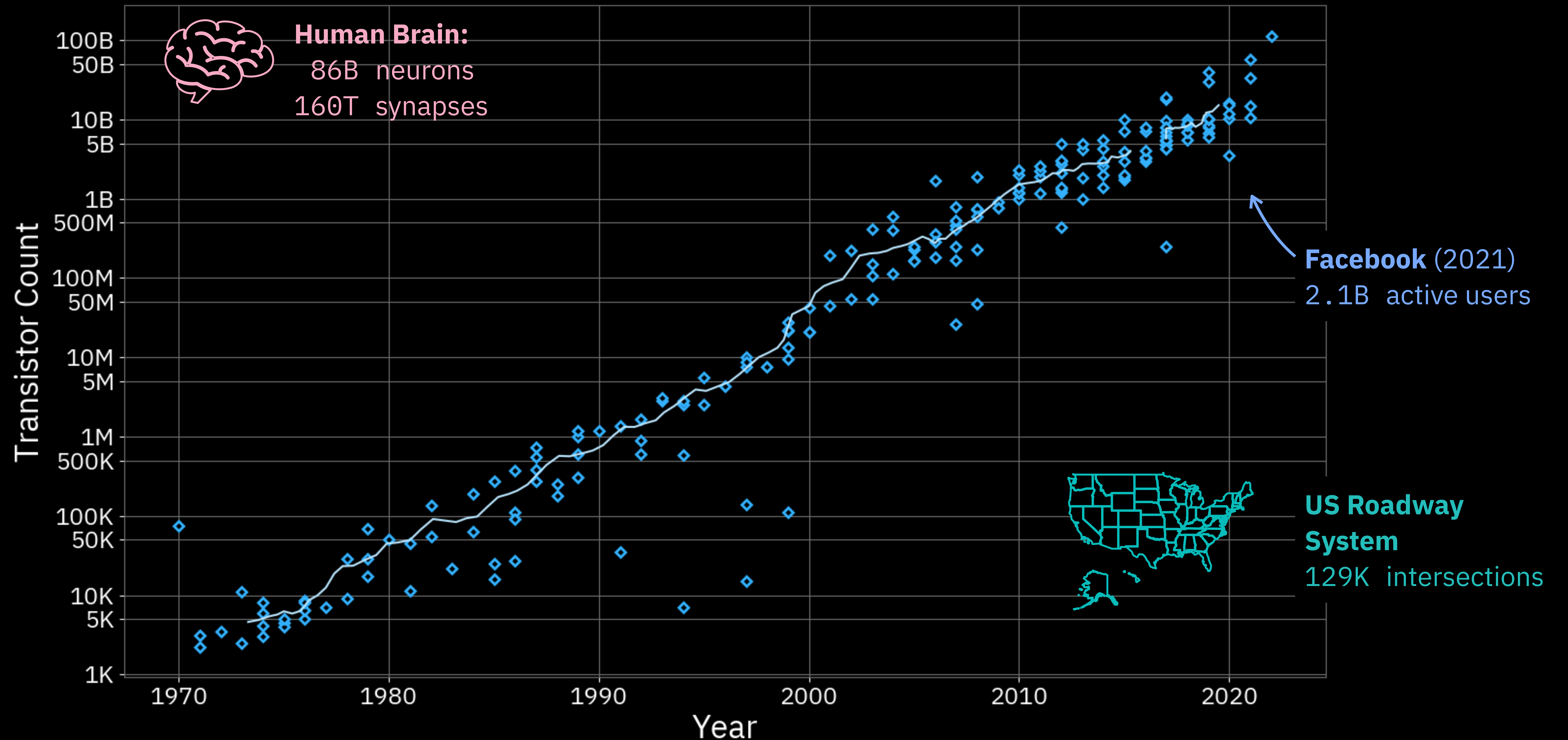


Design automation tools are required to build microprocessors!

Source: wikipedia.org/en/transistor_count

History of Transistor Count in Microprocessors

Moore's law: transistor count doubles every two years

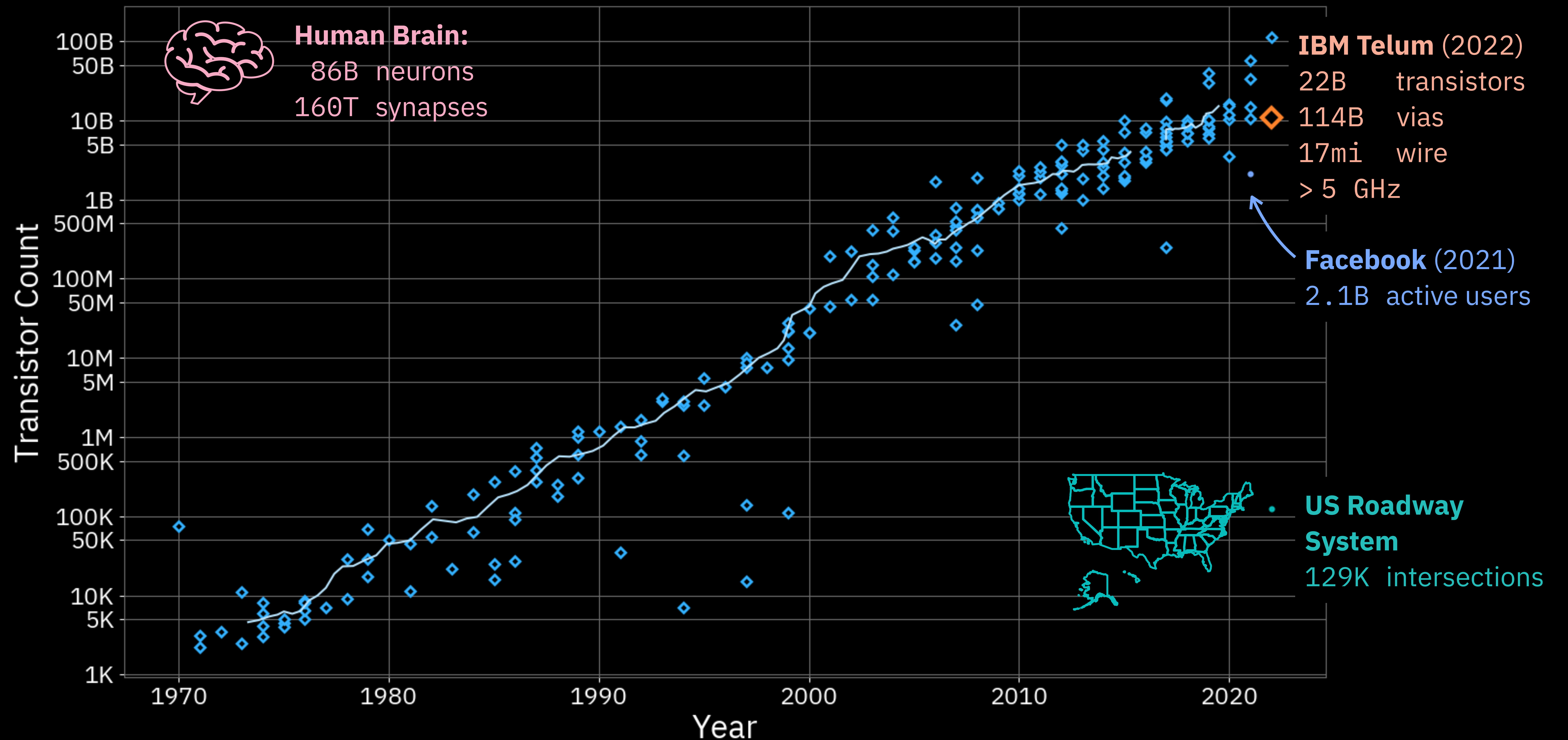


Design automation tools are required to build microprocessors!

Source: wikipedia.org/en/transistor_count

History of Transistor Count in Microprocessors

Moore's law: transistor count doubles every two years

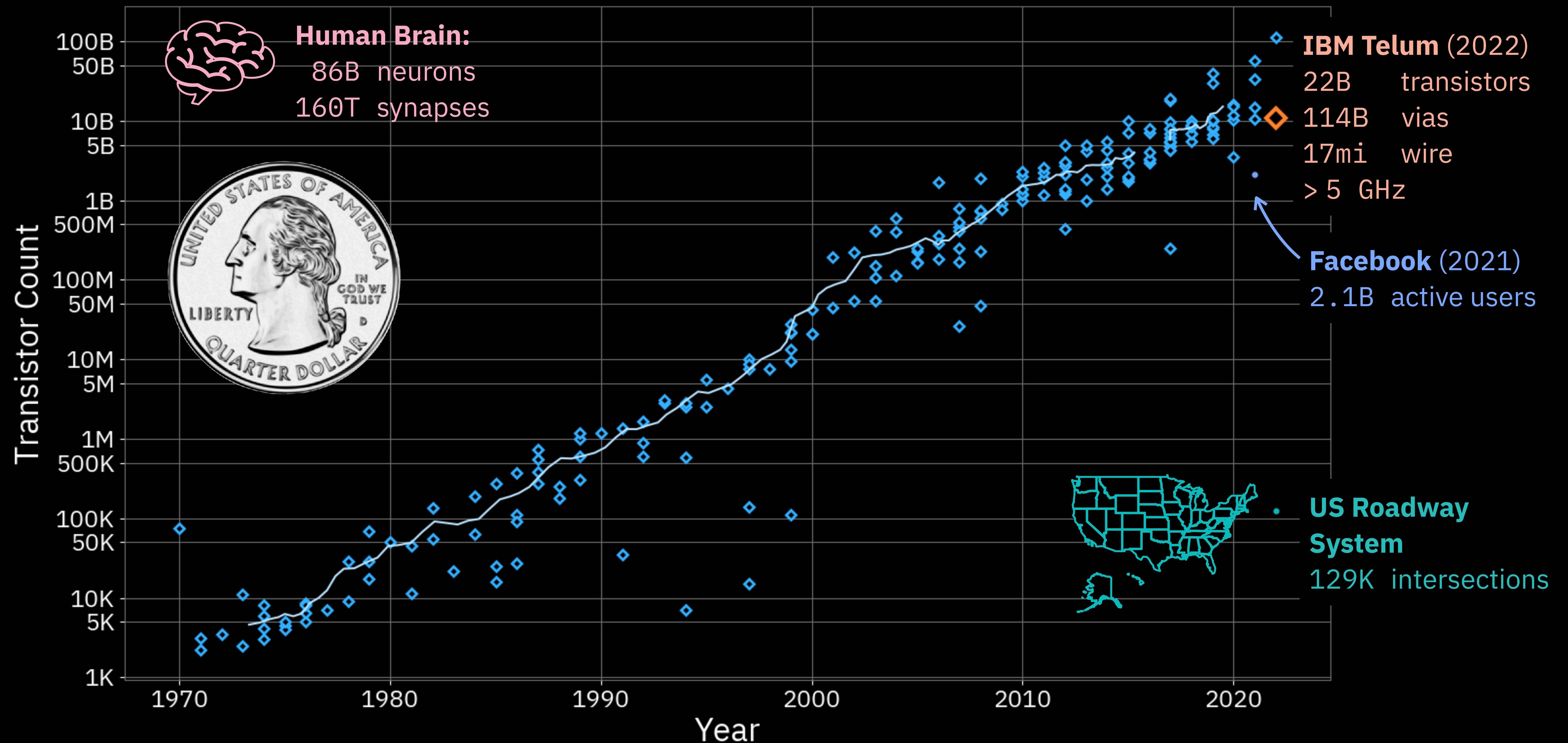


Design automation tools are required to build microprocessors!

Source: wikipedia.org/en/transistor_count

History of Transistor Count in Microprocessors

Moore's law: transistor count doubles every two years



Design automation tools are required to build microprocessors!

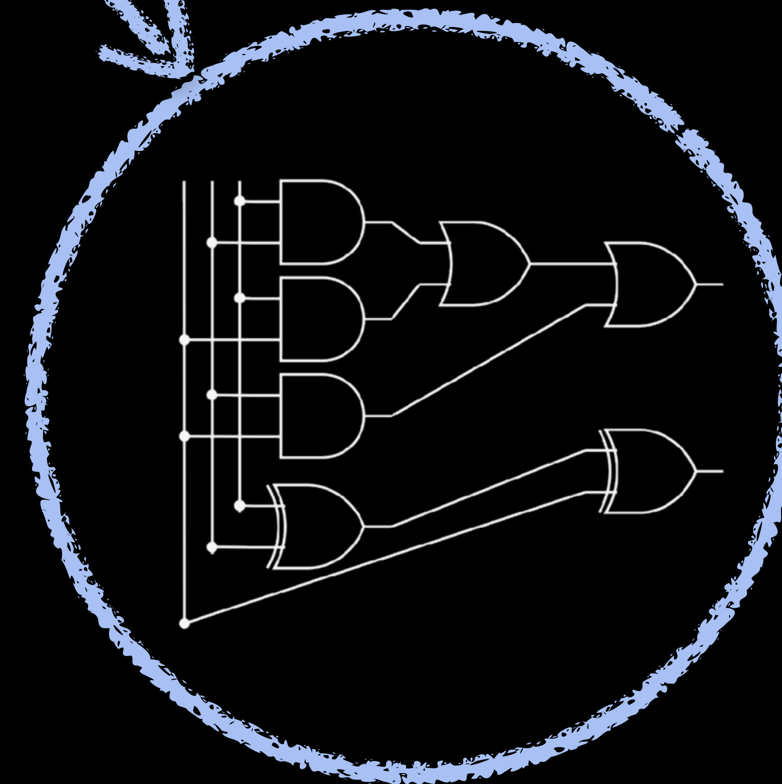
Source: wikipedia.org/en/transistor_count

Common Design Tasks

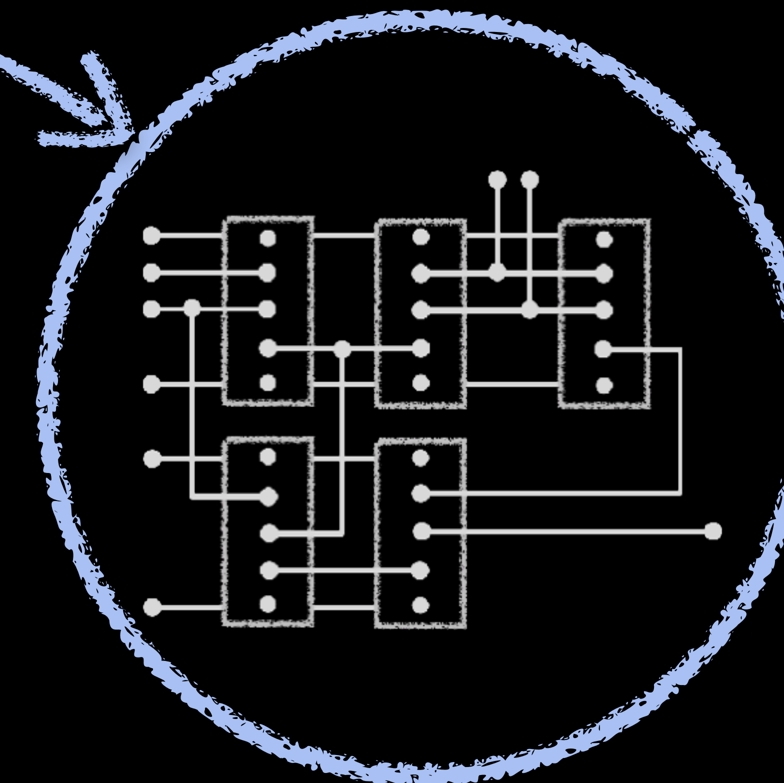
Define
Microarchitecture

A + B + Ci
=> S, Co

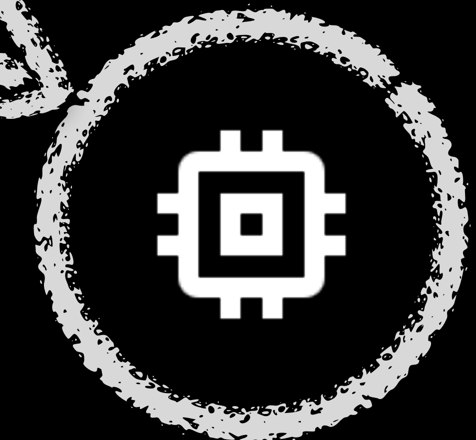
Logic
Description



Logic
Synthesis



Place &
Route



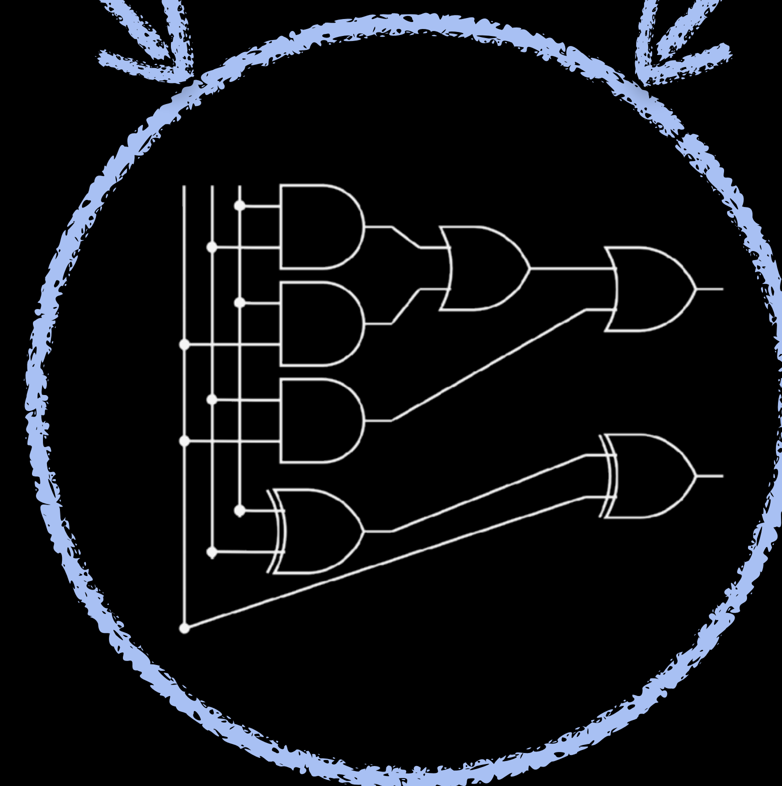
Microprocessor

Design Optimization: *An NP-Hard Problem*

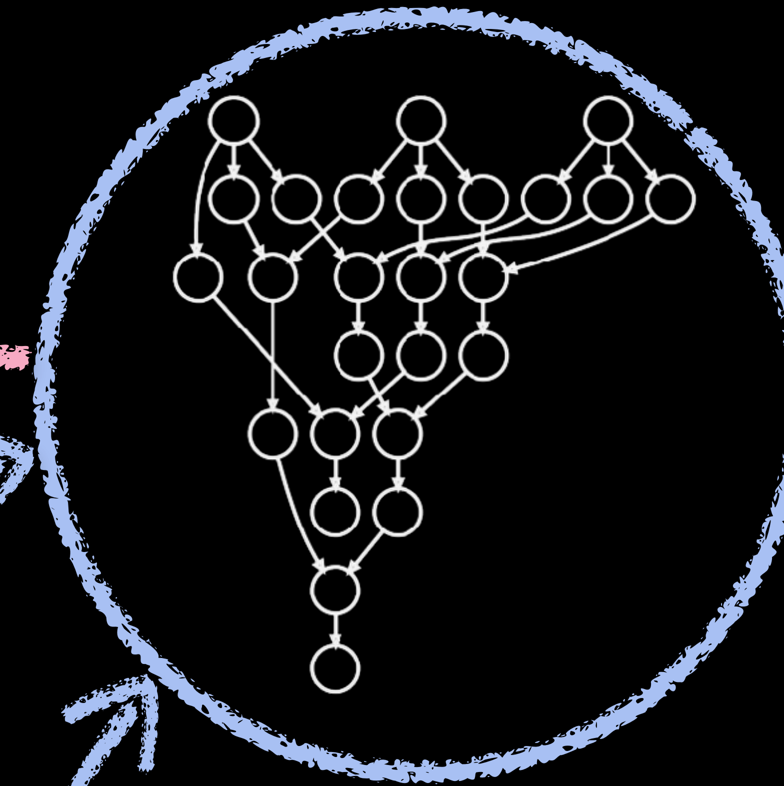
Define Microarchitecture

A + B + Ci
=> S, Co

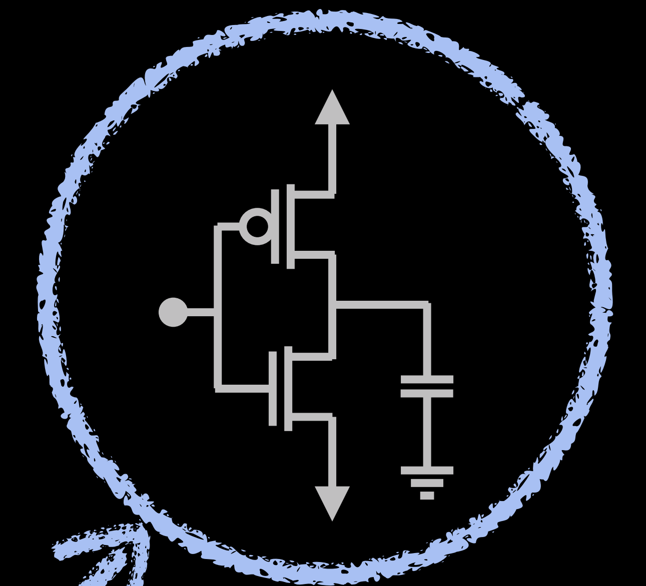
Logic Description



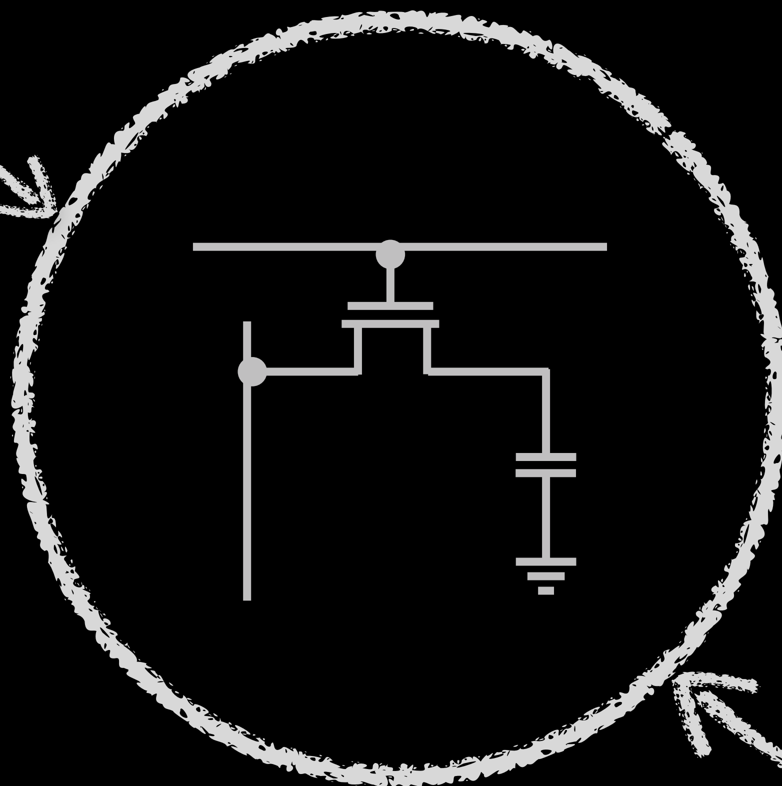
Logic Synthesis



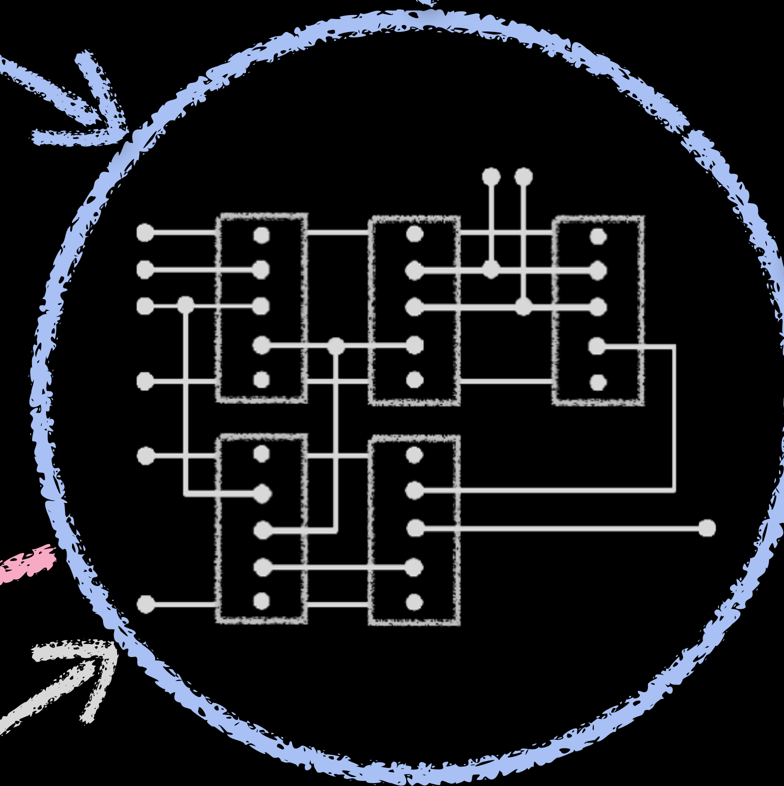
Timing Analysis



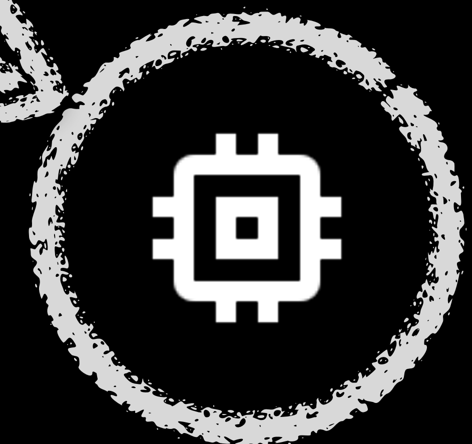
Power Analysis



Custom Circuits



Place & Route

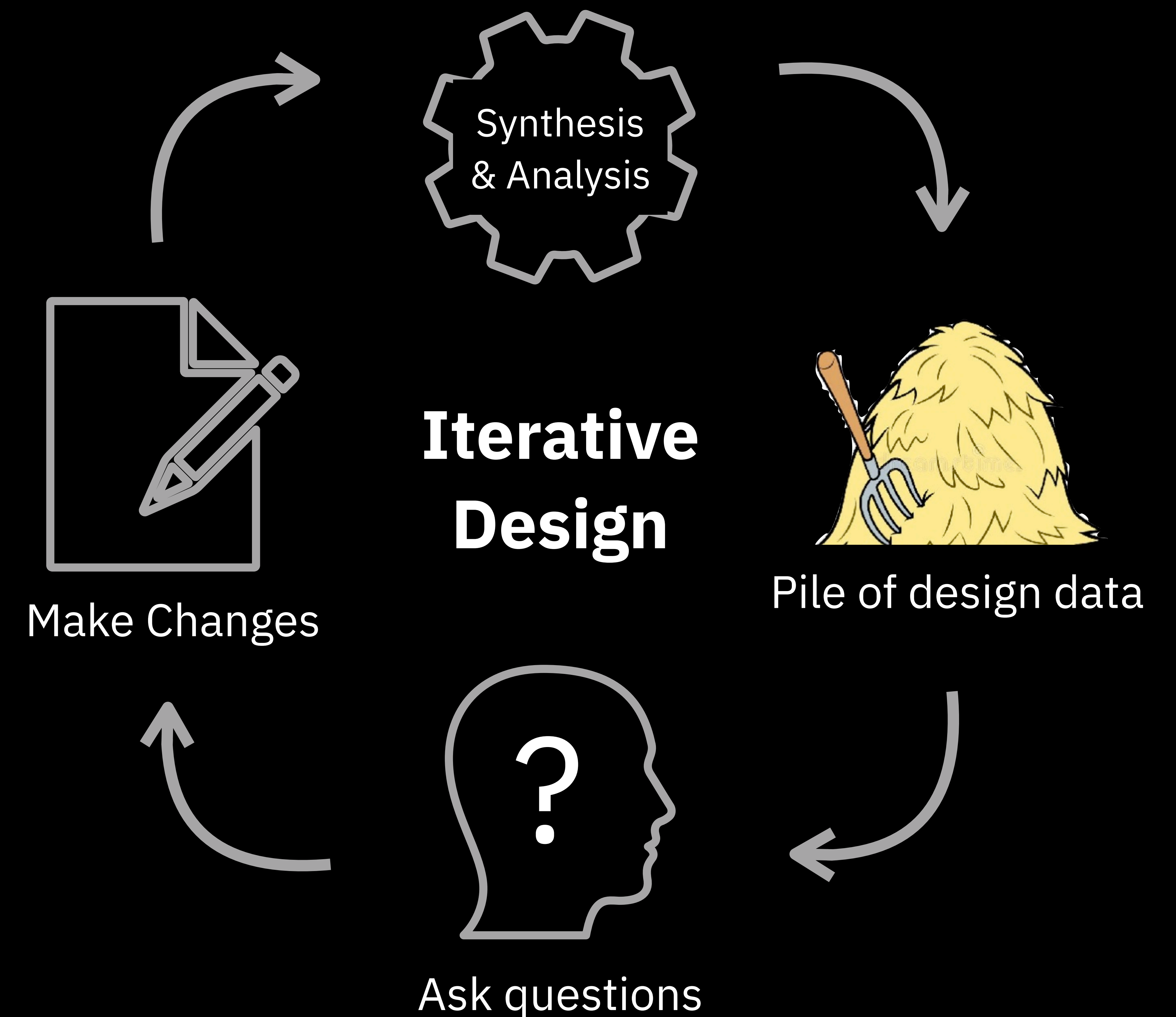


Microprocessor

The Problem – A haystack of design data

High-performance microprocessors are complicated devices.

- Processor design is an arduous and iterative process.
- Design automation is not simply "one and done."
- Questions are asked between each iteration:
    1. **What** happened?
    2. **Why** did it happen?
    3. **How** do we improve?

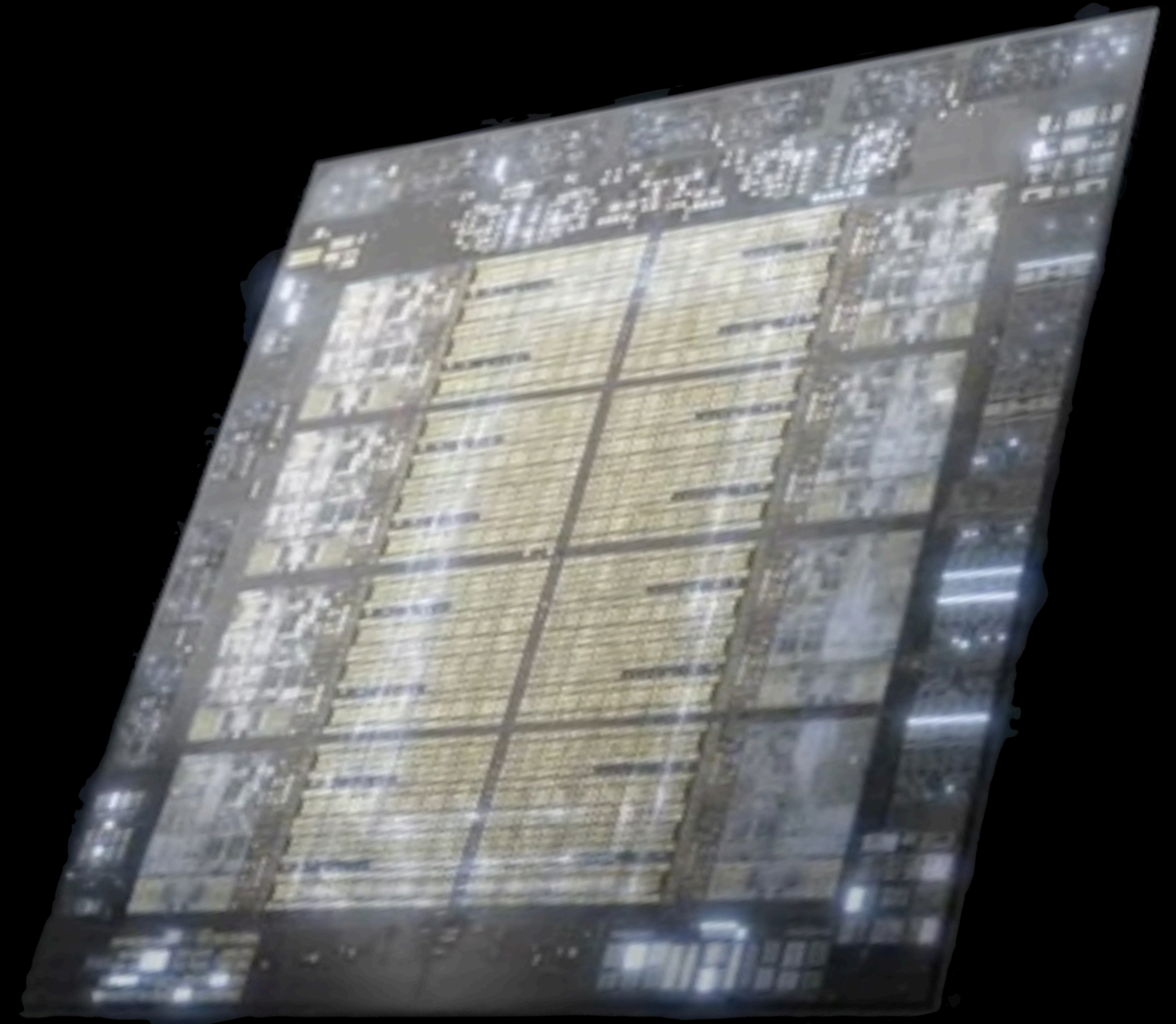


The Problem – A haystack of design data

High-performance microprocessors are complicated devices.

- Processor designs are separated into hierarchical components
- Each of these are analyzed separately and then stitched back together

2 Chips x 8 Cores x 9 Continents
= **A LOT OF DATA (41 GB*)**



IBM Telum Processor

*sum of size of compressed DD files on disk

Objectives

I.

How to build a microprocessor: **a Big Data Problem**

II.

An efficient **CPython** data model

III.

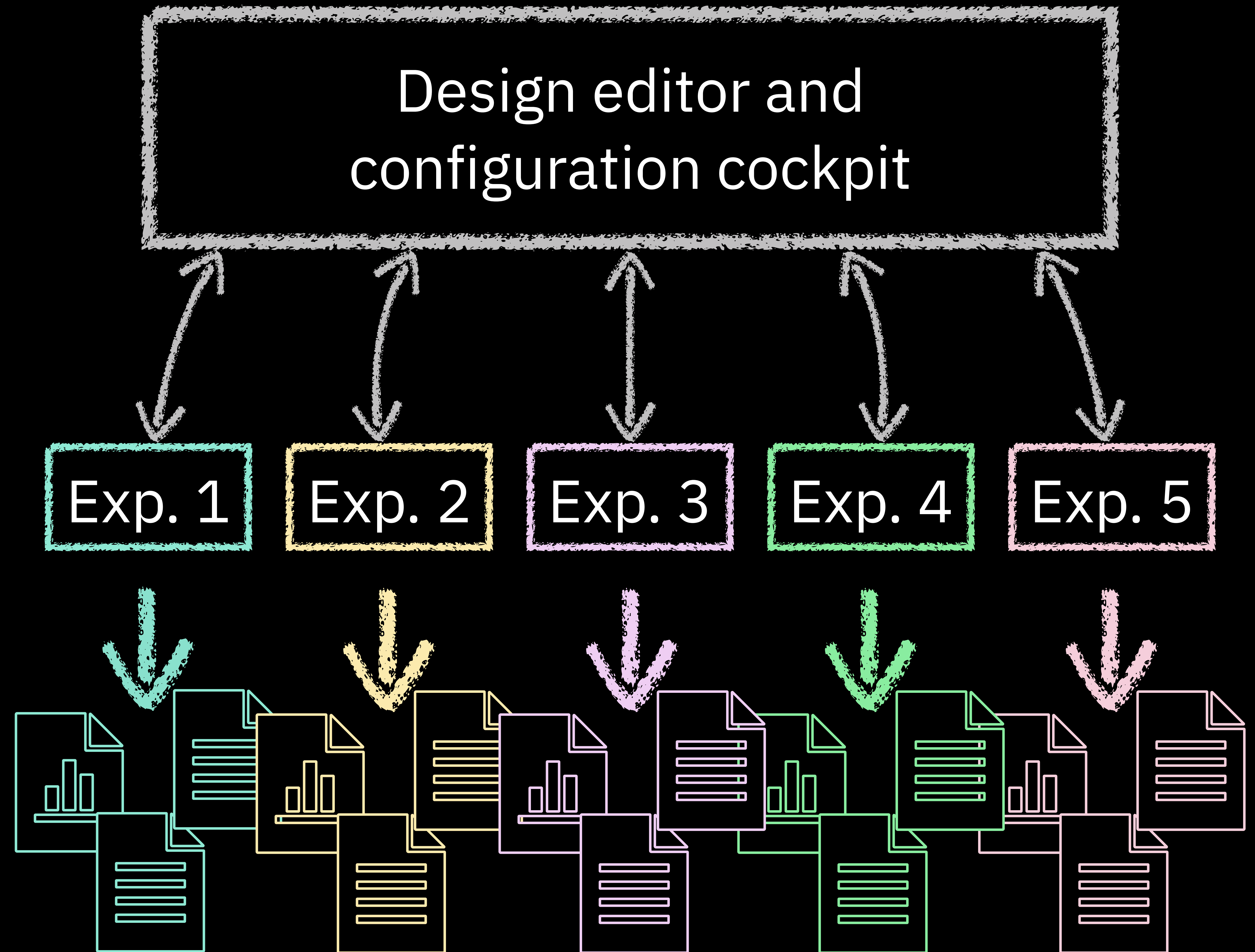
Using **Python** for data analysis

IV.

Lessons learned & Wrap-up

Key Dimensions

1. Managing design versions
2. Hierarchical components
3. Access to design and derived data
4. Team interlock and collaboration



EDA Application Layers

Design editor and configuration cockpit

Parms

Logic

Asserts

Spice

Distributed Batch Job Scheduler	RedHat Enterprise Linux (RHEL) OS	x86 | POWER

Distributed Clustered File System

EDA Application Layers

Design editor and configuration cockpit

Parms

Logic

Asserts

Spice

Optimization Engines

- Synthesis
- Placer
- Router
- Sign-Off

Analysis Engines

- Static Timing
- Dynamic Power
- Congestion
- Simulator

Distributed Batch Job Scheduler	RedHat Enterprise Linux (RHEL) OS	x86 | POWER

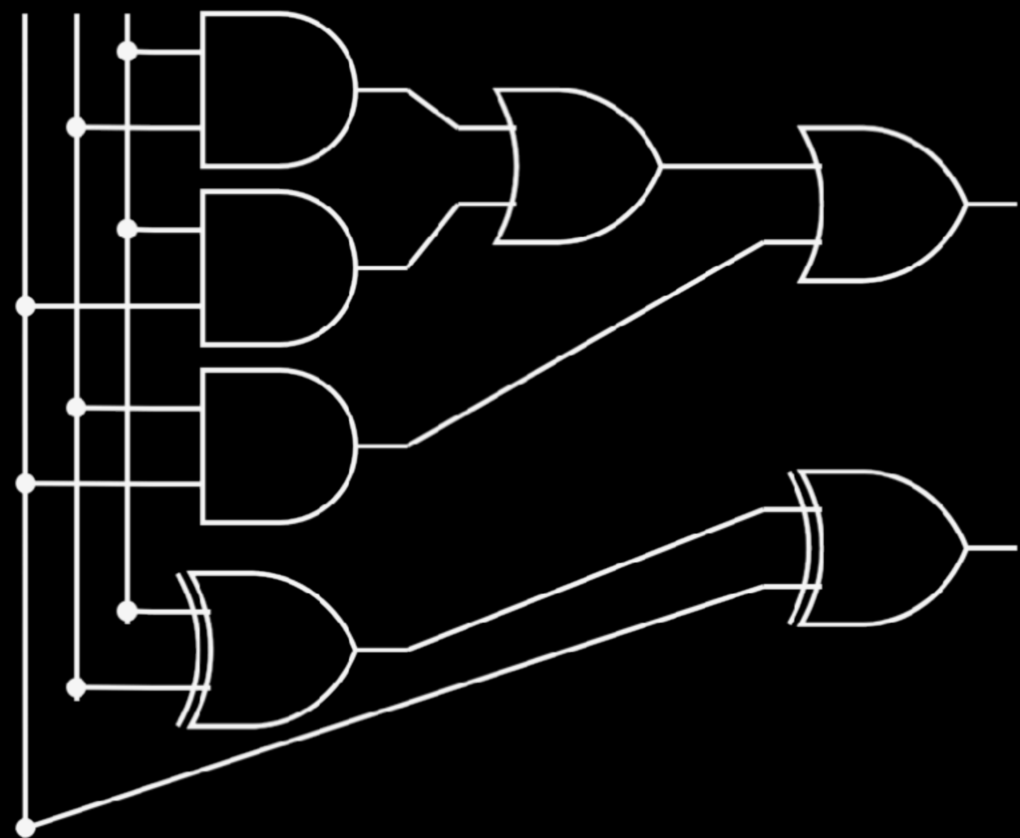
Distributed Clustered File System

EDA Application Layers

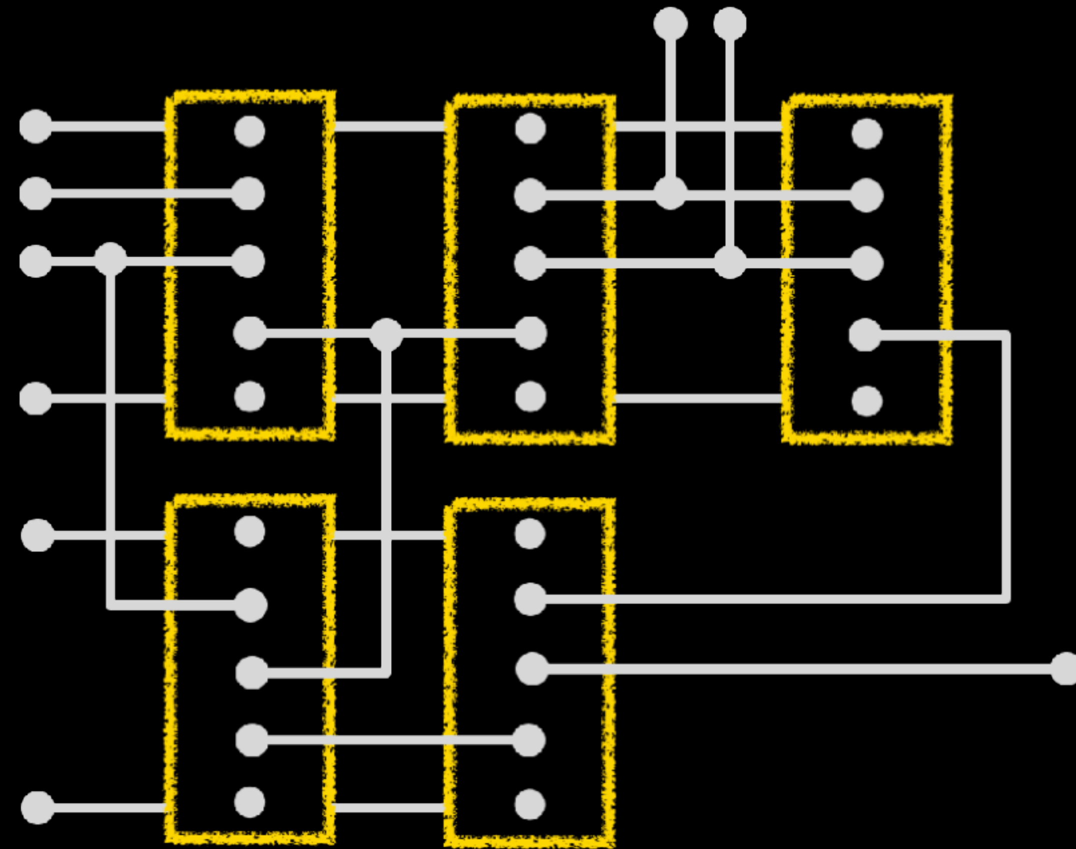
EDA Application Layers

Design Data (DD): A CPython binary data model and API

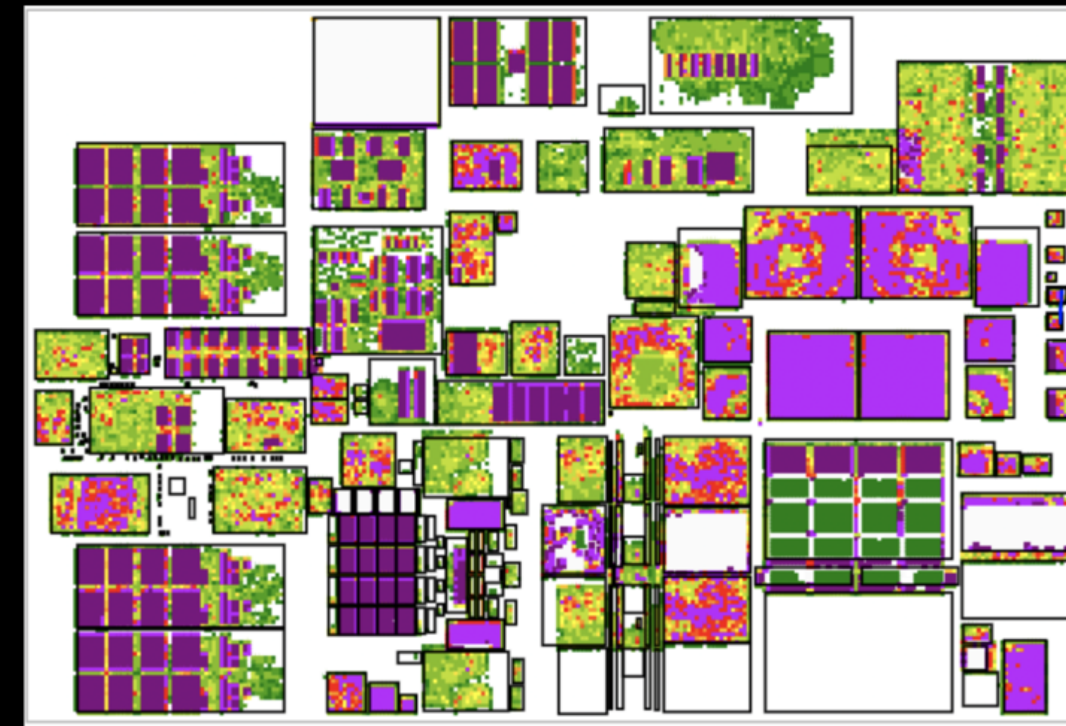
Logical "Netlist"



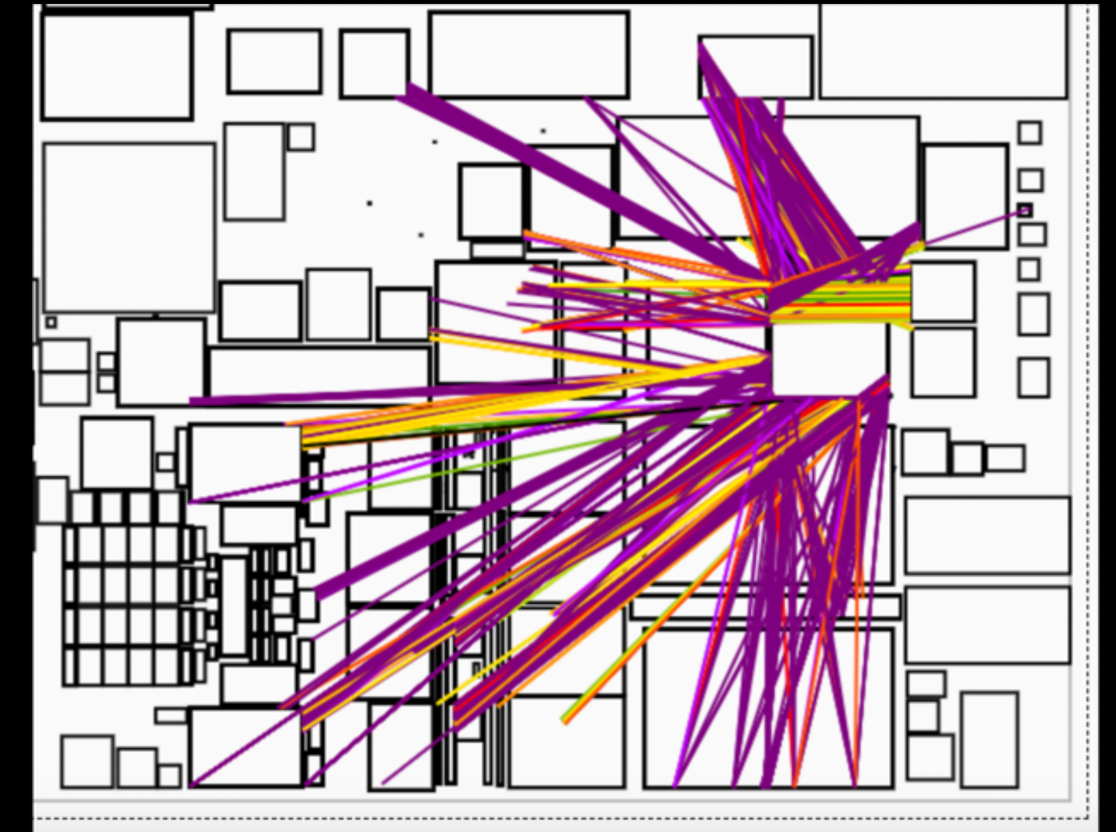
Placement & Wires



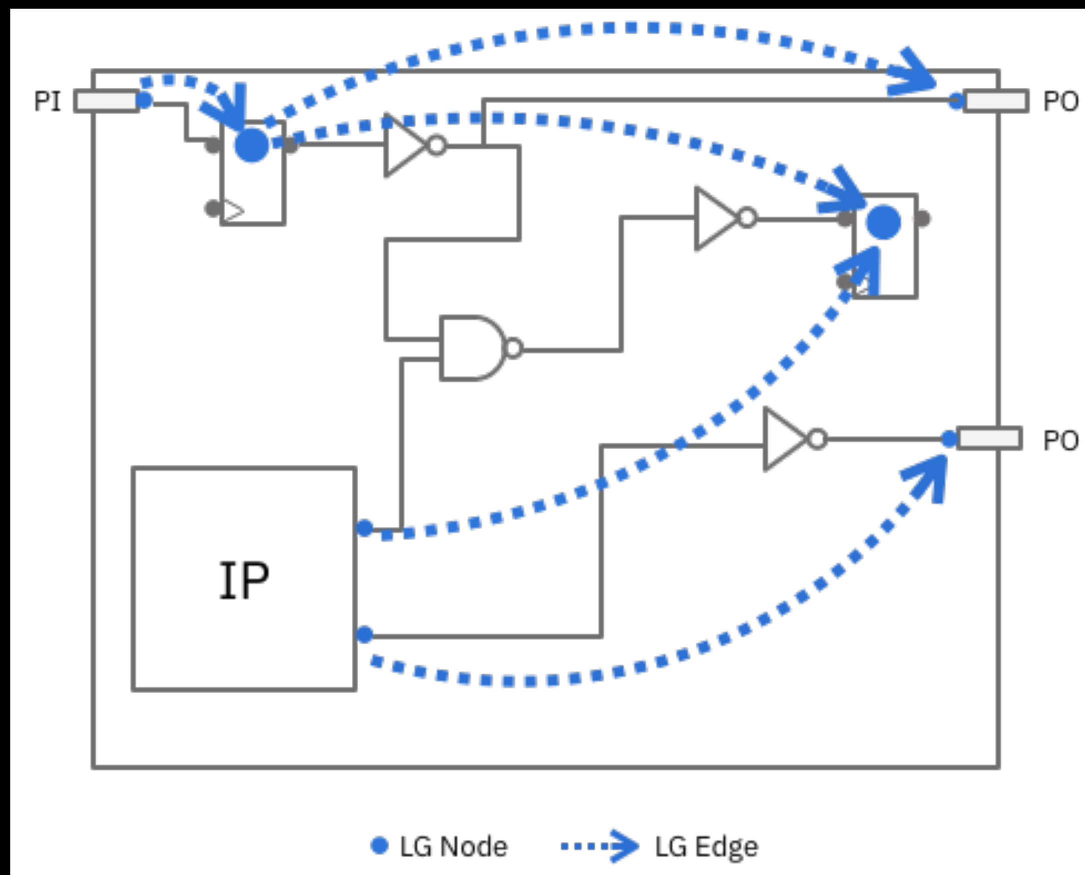
Placement Density



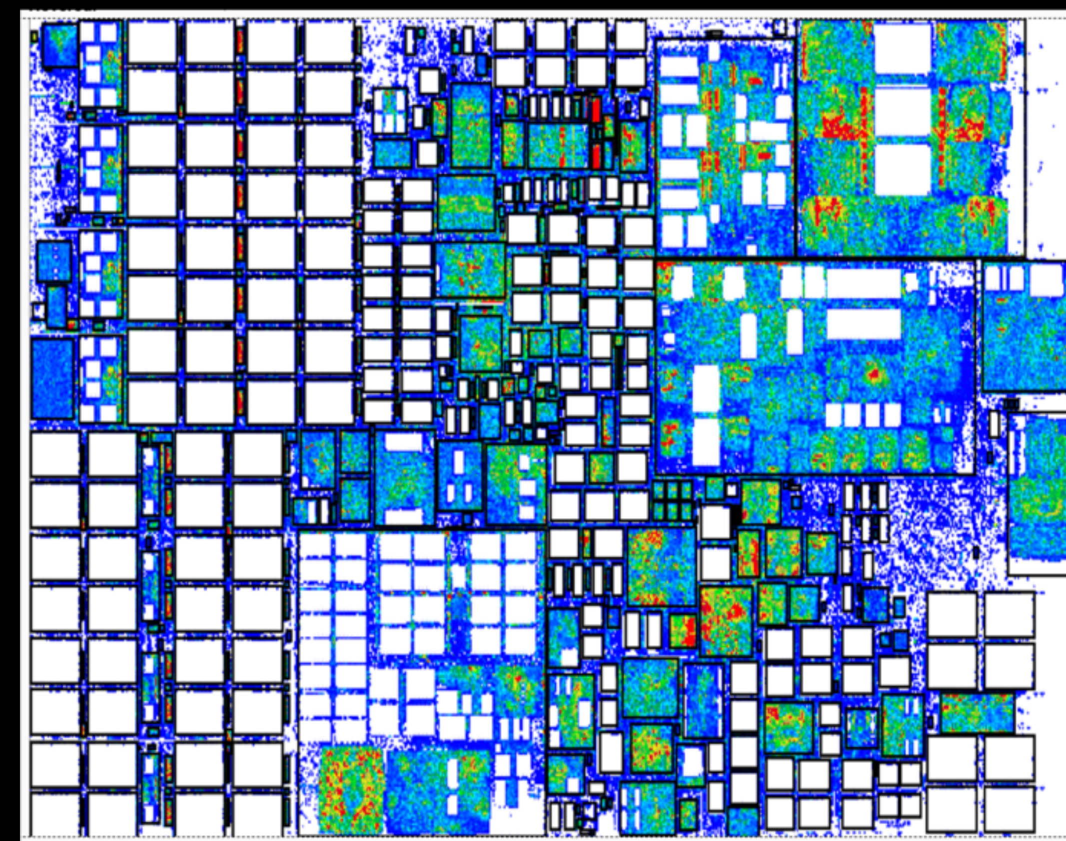
Floorplan forces



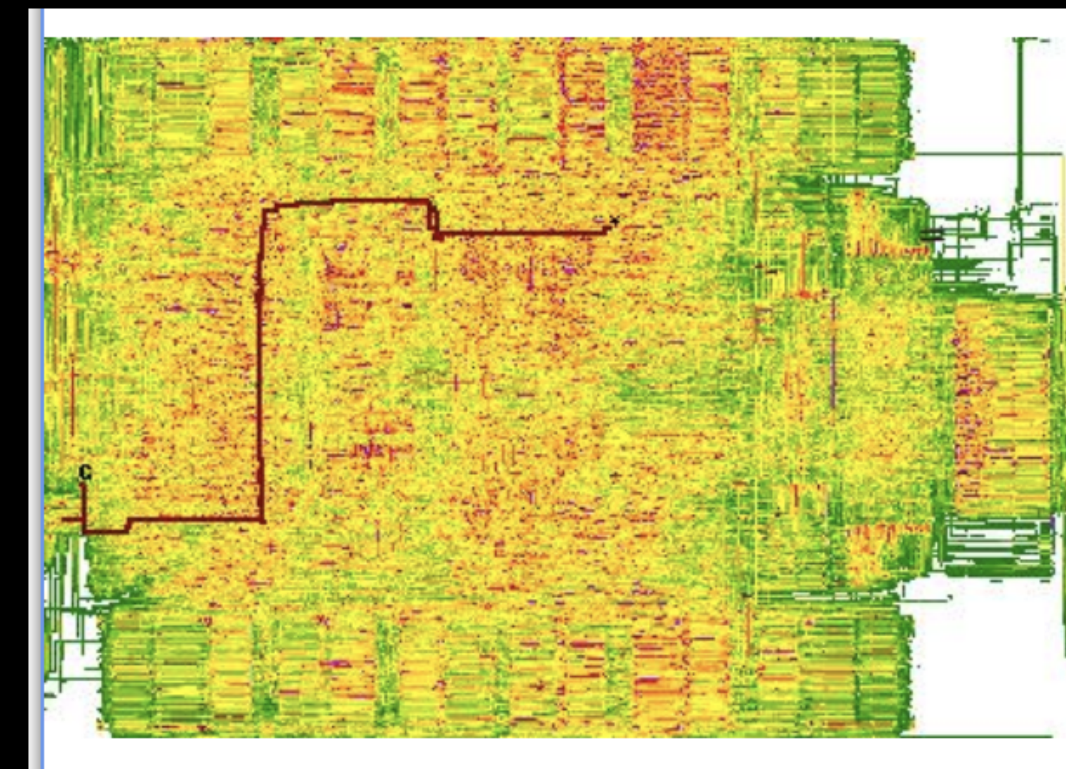
Signal Graph



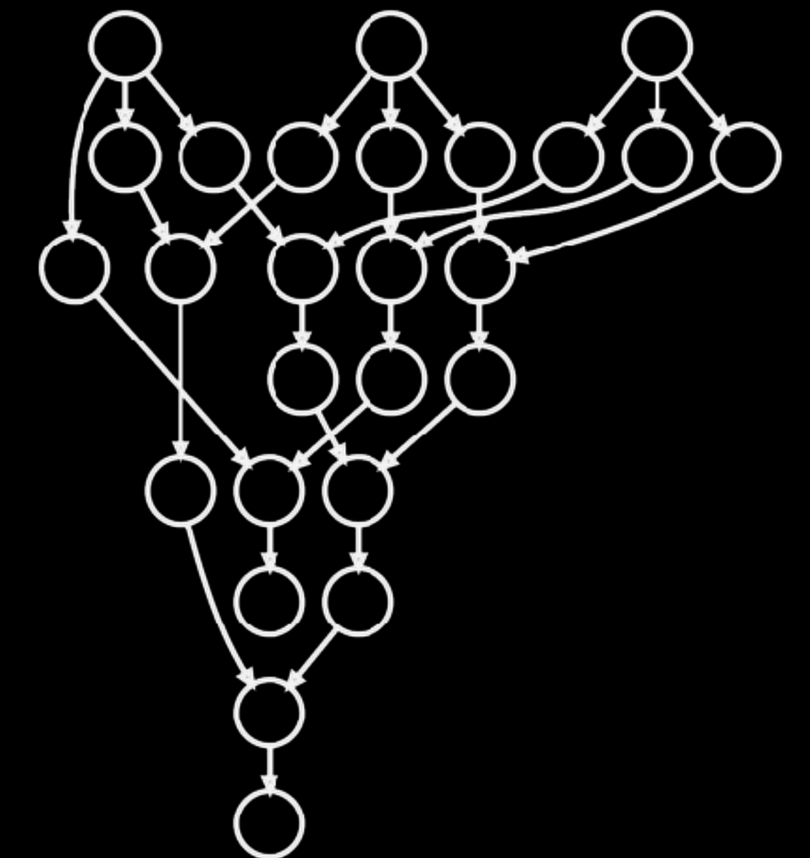
Leakage Density



Routing Congestion

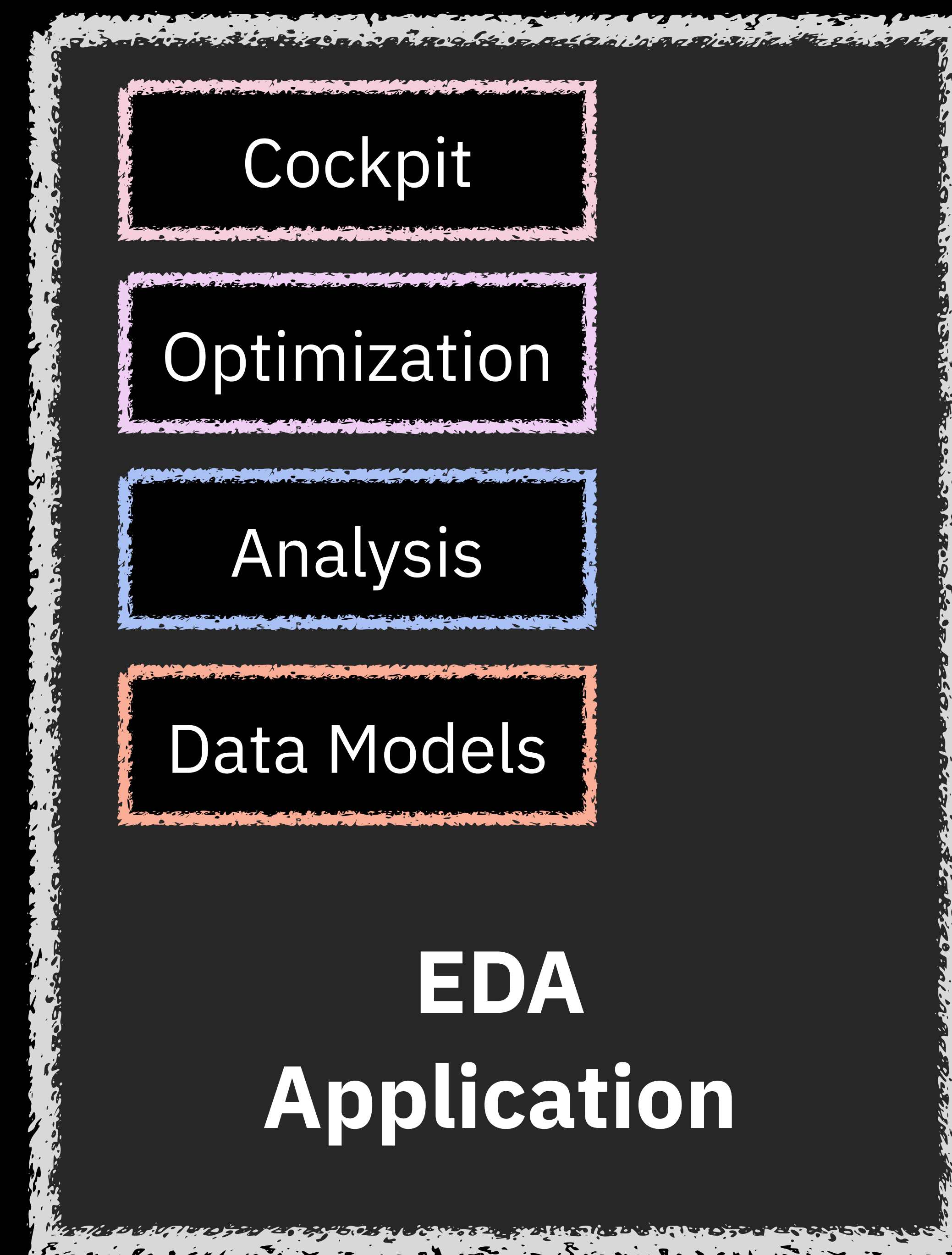


Timing



DD is a read-only, self-contained, binary file database

Design Data (DD): A CPython binary data model and API



16 CPU • 600 GB • 8 hrs

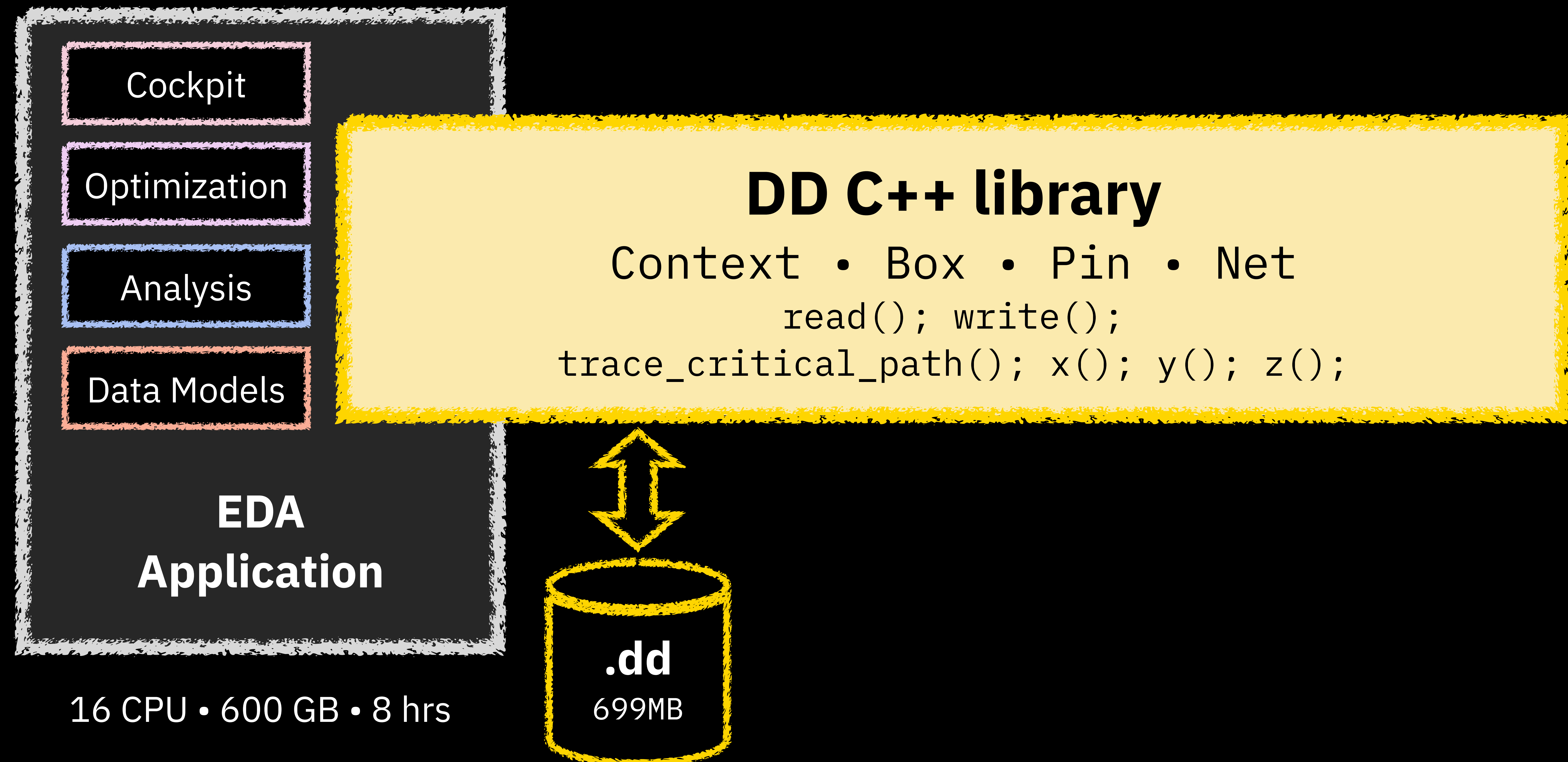
Design Data (DD): A CPython binary data model and API

Benefits

- Smaller memory footprint and faster compute performance
- Custom memory management and object initialization
- Multithreading
- Support multiple execution environments

Drawbacks

- Maintain custom Python objects and iterators
- Extra layer creates additional complexity and maintenance.
- Execution outside of Global Interpreter Lock (GIL)



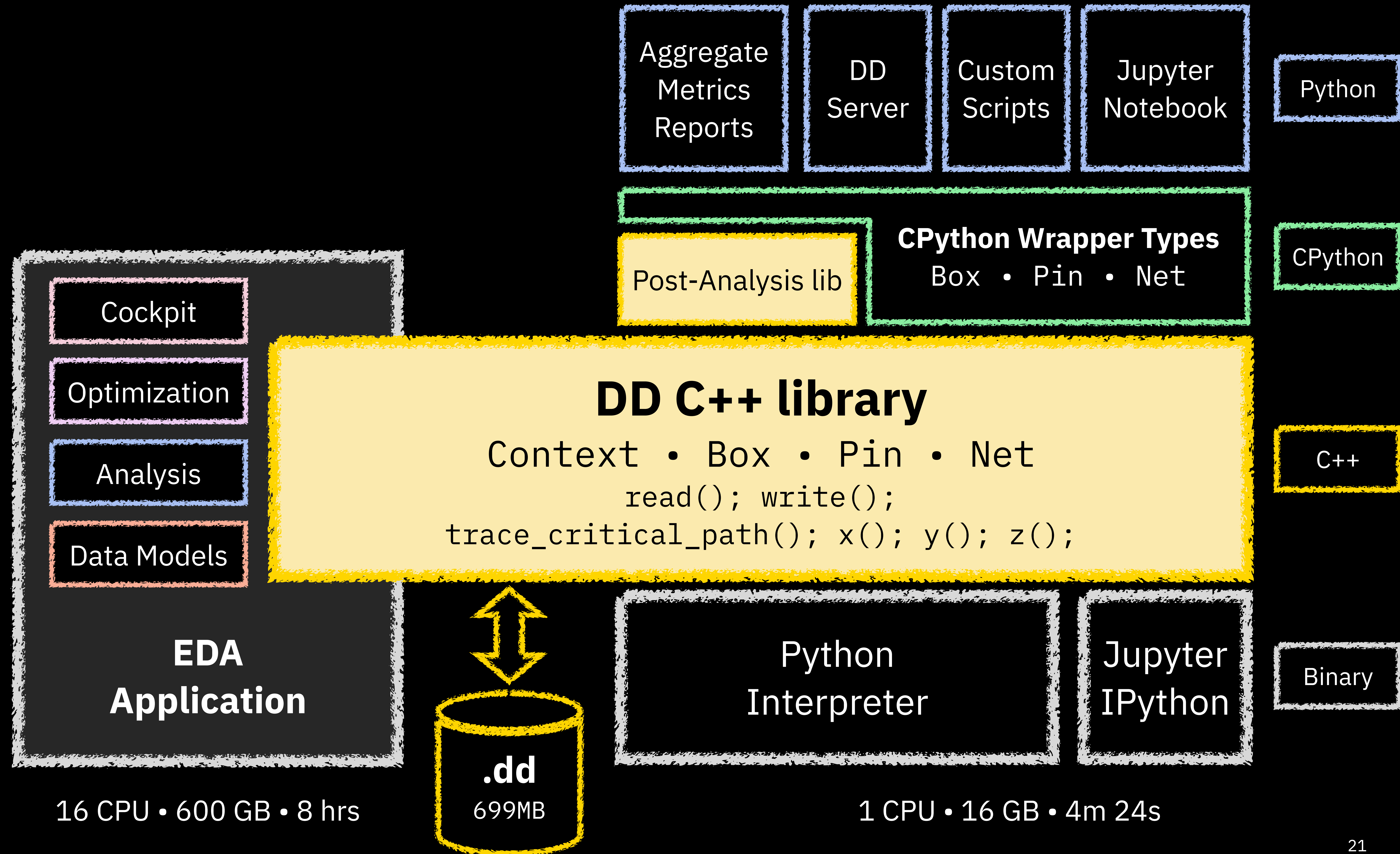
Design Data (DD): A CPython binary data model and API

Benefits

- Smaller memory footprint and faster compute performance
- Custom memory management and object initialization
- Multithreading
- Support multiple execution environments

Drawbacks

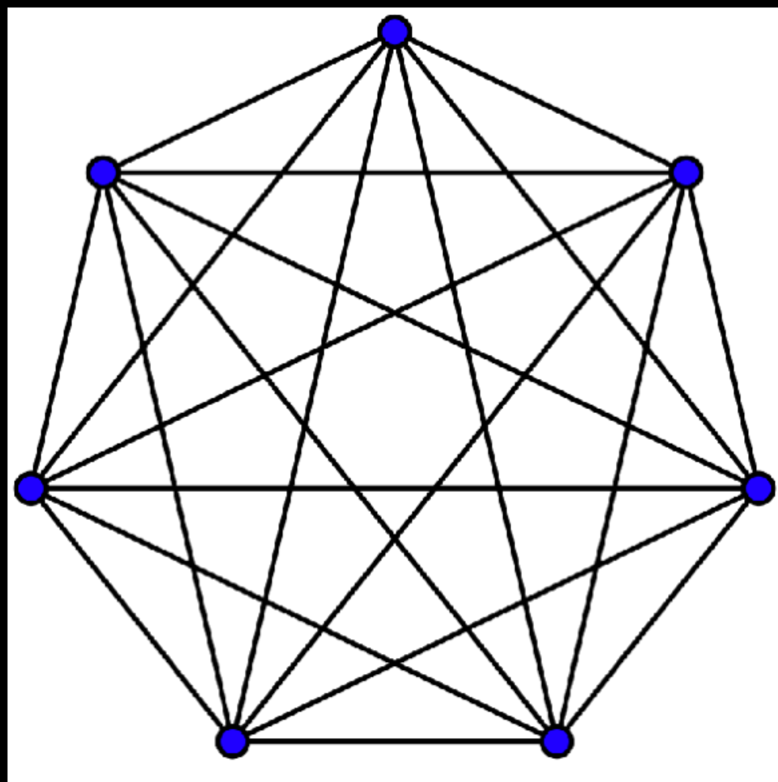
- Maintain custom Python objects and iterators
- Extra layer creates additional complexity and maintenance.
- Execution outside of Global Interpreter Lock (GIL)



Python vs. C++

CPython provides the best of both worlds!

Complete graph data model performance



```
Create a complete graph of:
  10,000   vertices
  49.99 M  edges

Python:     6 min, 8.1 GB
C++:     3.45 sec, 1.2 GB
```

Python

- Rapid App Development (i.e., fast prototyping)
- Dynamic and flexible
- Large community for package development and support
- Support C++ integrations for performance

C++

- Fast and memory efficient
- Strong typing
- Multi-threading

Use **Python packages** for data analysis and management:

- `pandas (DataFrame)`
- `matplotlib (pyplot)`
- `websockets & asyncio`
- `flask (web server)`
- `tensorflow`
- `scikit-learn`
- `DB connectors`
- `PIL (ImageDraw)`
- `jupyter`

Objectives

I.

How to build a microprocessor: **a Big Data Problem**

II.

An efficient **CPython** data model

III.

Using **Python** for data analysis

IV.

Lessons learned & Wrap-up

Use Models

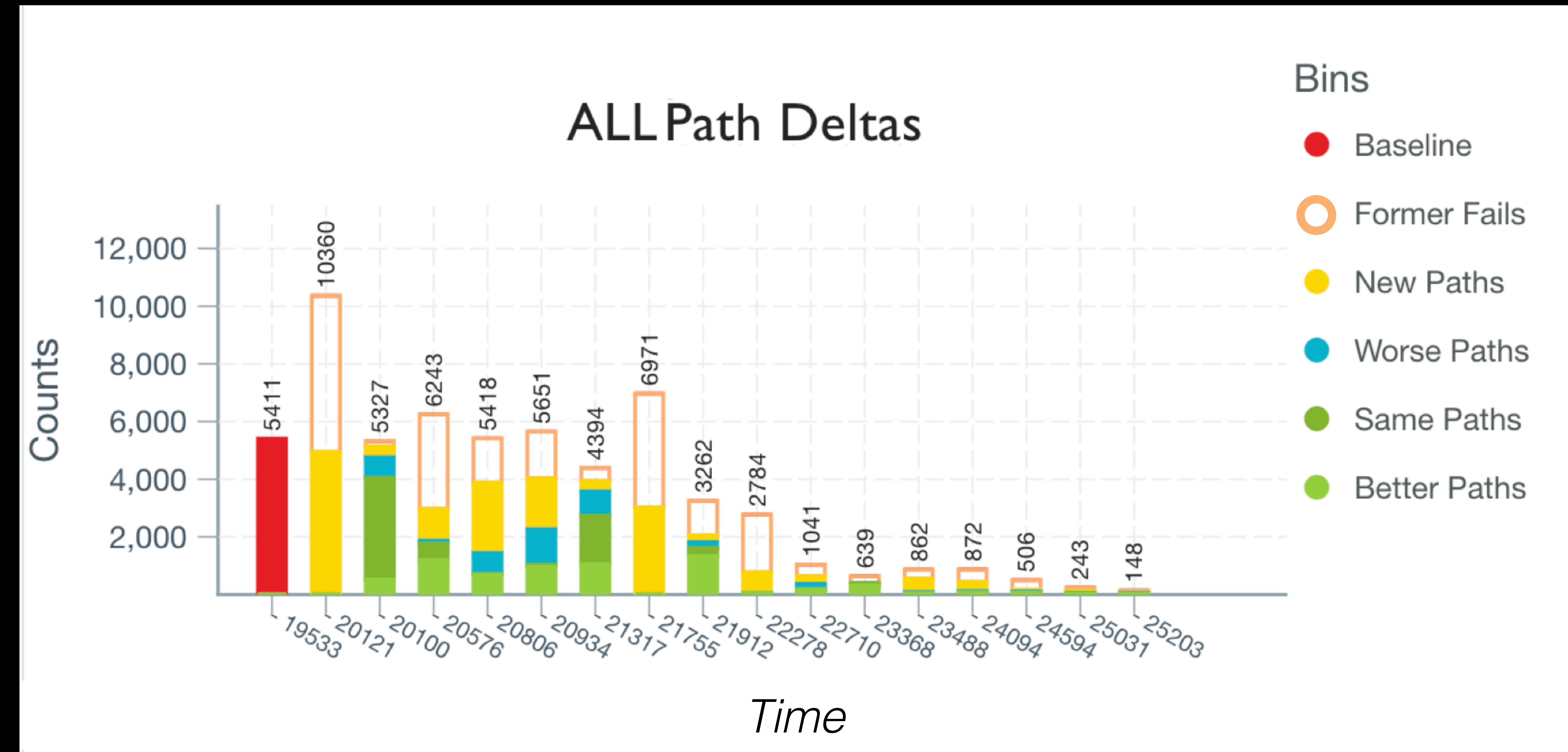
Users and developers may access data in whichever form helps them accomplish their current task most effectively.

Use C++ extension modules for runtime performance

Track progress over time

via a flask web server and pandas

```
import flask, flask_restful, pandas
class TakedownPathDeltas(flask_restful.Resource):
    def get(self, args):
        hists = dict()
        for run_id in self.get_runs_list(args):
            run_df = DataFrame(db_conn.query(
                f'SELECT slack FROM Cache_{run_id};'))
            bins = pandas.cut(run['slack'], self.bins)
            hists[run_id] = run_df.groupby(bins)['slack']
                            .count().to_dict()
        return hists


app = flask.Flask(__name__)
api = flask_restful.Api()
api.add_resource(TakedownPathDeltas,
                 '/api/takedown/path_deltas')
```

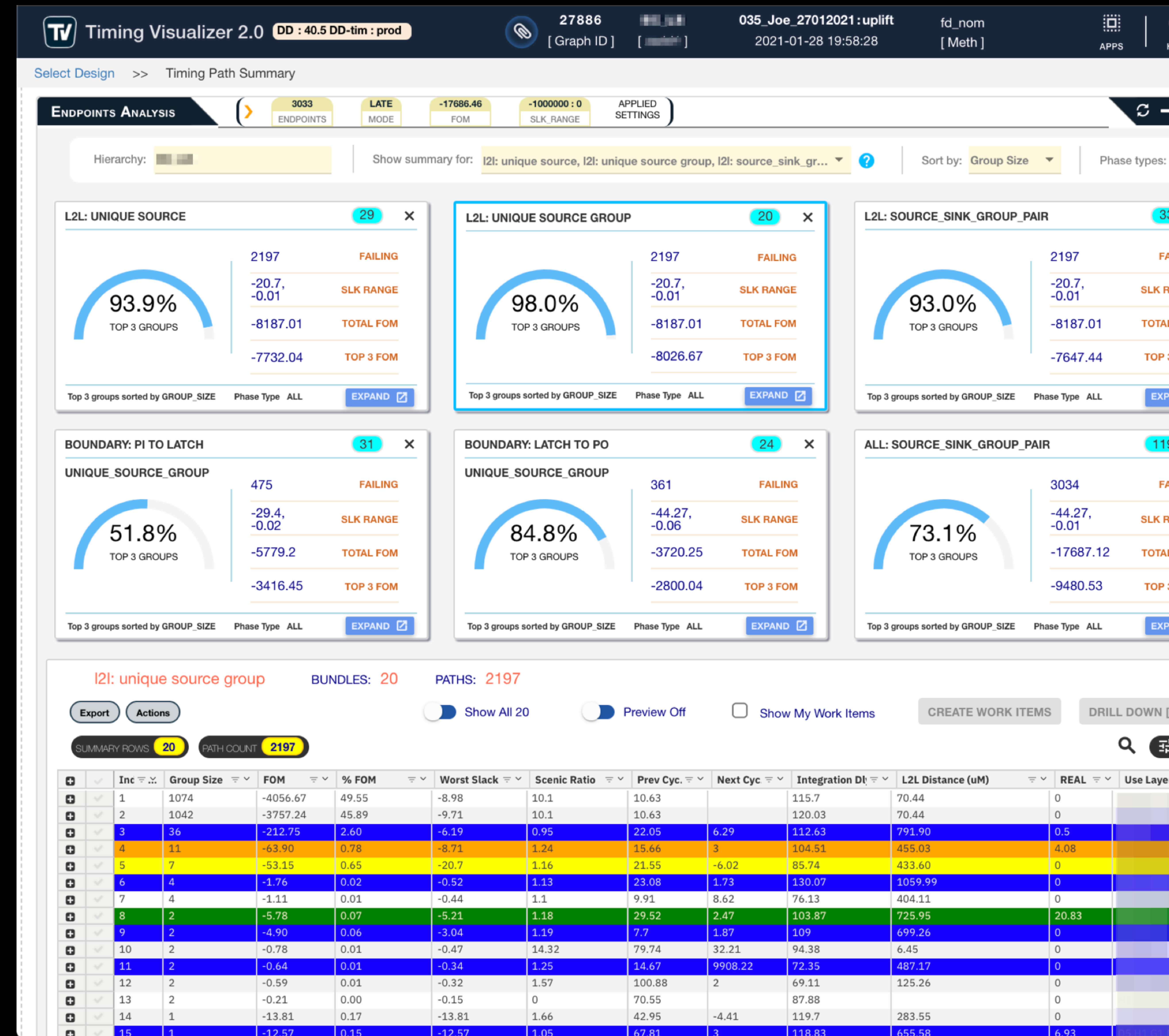


Goal: Form a mental model of the "whole picture"

Full Hierarchical Summary

via a websocket server

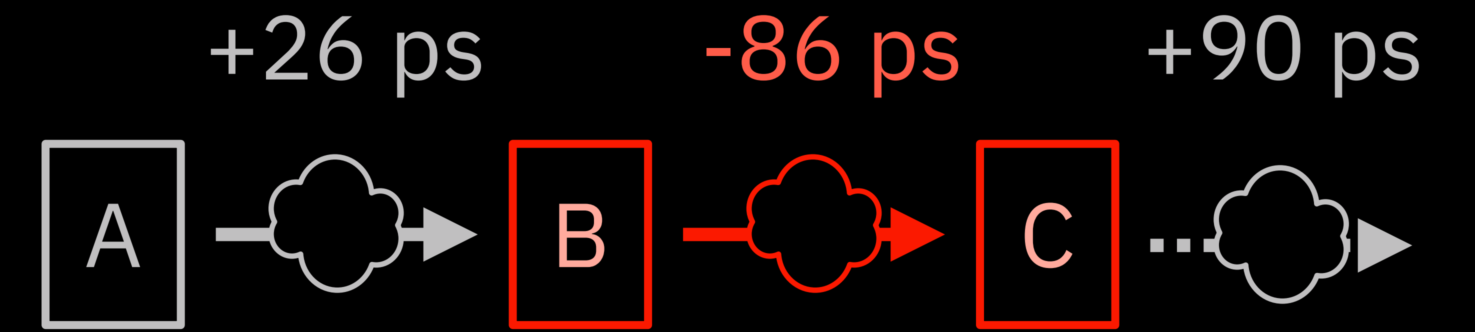
```
import asyncio, websockets
def aggregate_path_data(name):
    path_d = defaultdict(float)
    tpt = ctx.root_def().locate_pin(name)
    for t in tpt.iterate_critical_trace_in():
        if t.is_gate:
           path_d['gate_delay']  += t.delay()
        elif t.is_wire:
            path_d['wire_delay'] += t.delay()
    return path_d

async def handle_msg(conn, path):
    async for msg_d in conn:
        try:
            res_d = aggregate_path_data(msg_d[‘name’])
            conn.send(res_d)
        except Exception as e:
            conn.send(json.dumps({'error': e}))


ctx = dd.read(DD_FILE)
start_server = websockets.serve(handle_msg, hostname, port)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```



Query **DD data** and compute aggregate metrics.

How do I fix it?

Multi-cycle, multi-hierarchy path example



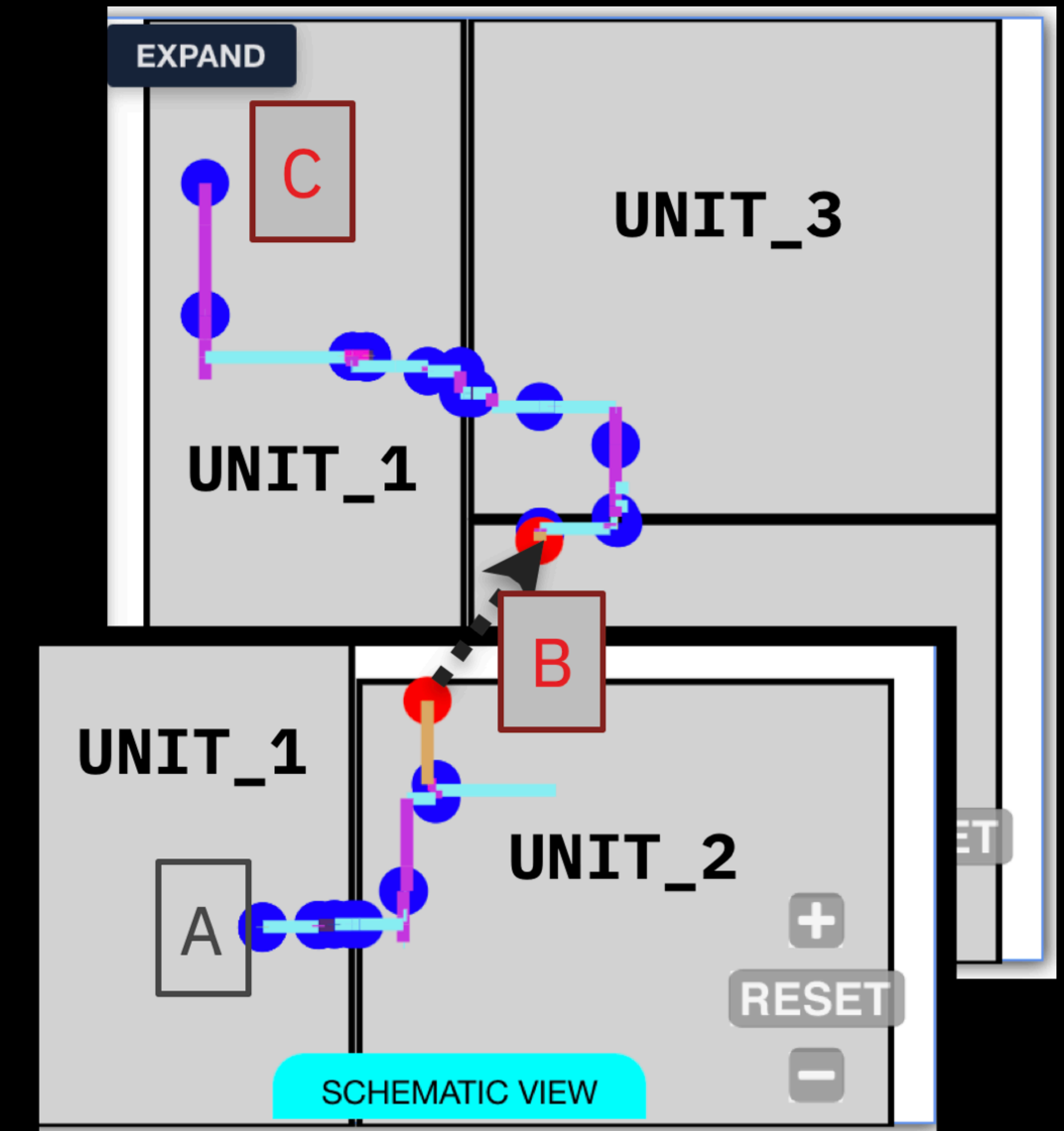
All: source_sink_group_pair BUNDLES: 396 PATHS: 36041

Export Actions Show All 53 Preview Off Show My Work Items CREATE WORK ITEMS DRILL DOWN [###1]

SUMMARY ROWS 53 PATH COUNT 217 FILTERS >>> Prev Cyc. > 0 Next Cyc. > 40 CLEAR ALL

	Group	FOM	% FOM	Worst Slack	Prev C	Next	Units	Defs	EndDef	Task	Phase
32	48	-3598.67	0.33	-85.87	25.93	9817.71	[illegible]	[illegible]	[illegible]	Custom Circuit	M@L

```
def render_path_layout(msg_d):
    pins, wires = (list(), list())
    tpt = ctx.locate_pin(msg_d['name'])
    for t in tpt.iterate_critical_trace_in():
        pins.append({ "coords": (t.pin().x(), t.pin().y(), t.pin().z()), })
        for wire in t.net().iterate_wires():
            wires.append({
              "start": [*wire.start().coords()],
              "end":   [*wire.end().coords()]
            })
    return {"pins": pins, "wires": wires}
```

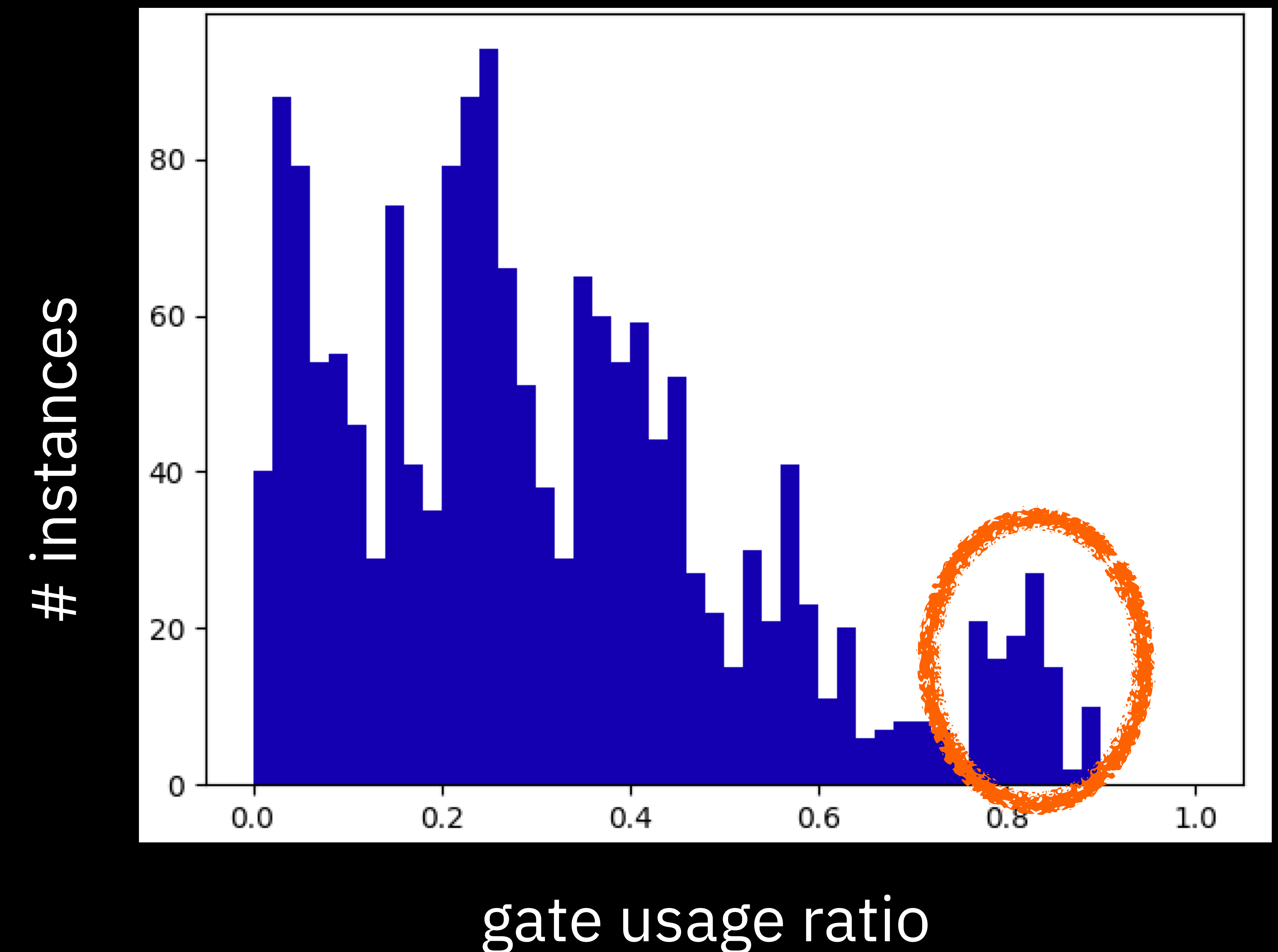


Multi-latch and multi-hierarchy path

Load and stitch all data files generated from ***separate hierarchical components*** and render path coordinates.

DD, which failing data paths have slow devices?

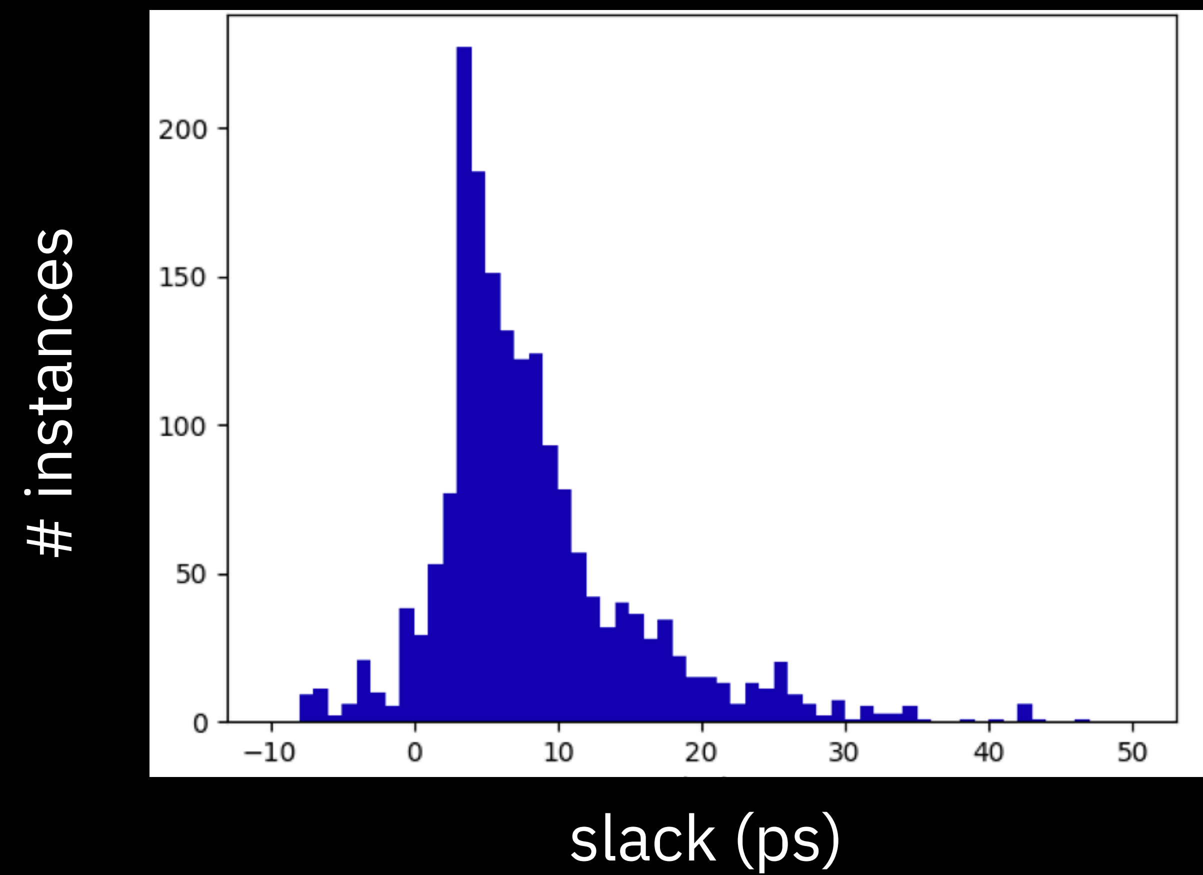
```
0   c = dd.read(DD_FILE)
1   epts = [e for e in c.root_def().iterate_end_points()
2           if e.worst_slack().slack() < 0]
3   for e in epts:
4       nboxes = 0
5       path_vt = defaultdict(int)
6       for ti in e.iterate_critical_trace_in():
7           if ti.box() ≠ None:  nboxes+=1
8           if ti.vt()  ≠ “":    path_vt[ti.vt()] +=1
9       traces.append({
10        **{“n": nboxes, "ept": e.name() },
11        **path_vt,
12        **{k+’%’: v / nboxes for k,v in path_vt.items()},
13      })
14  df = DataFrame(traces)
15  df[‘slow_%’].hist()
```

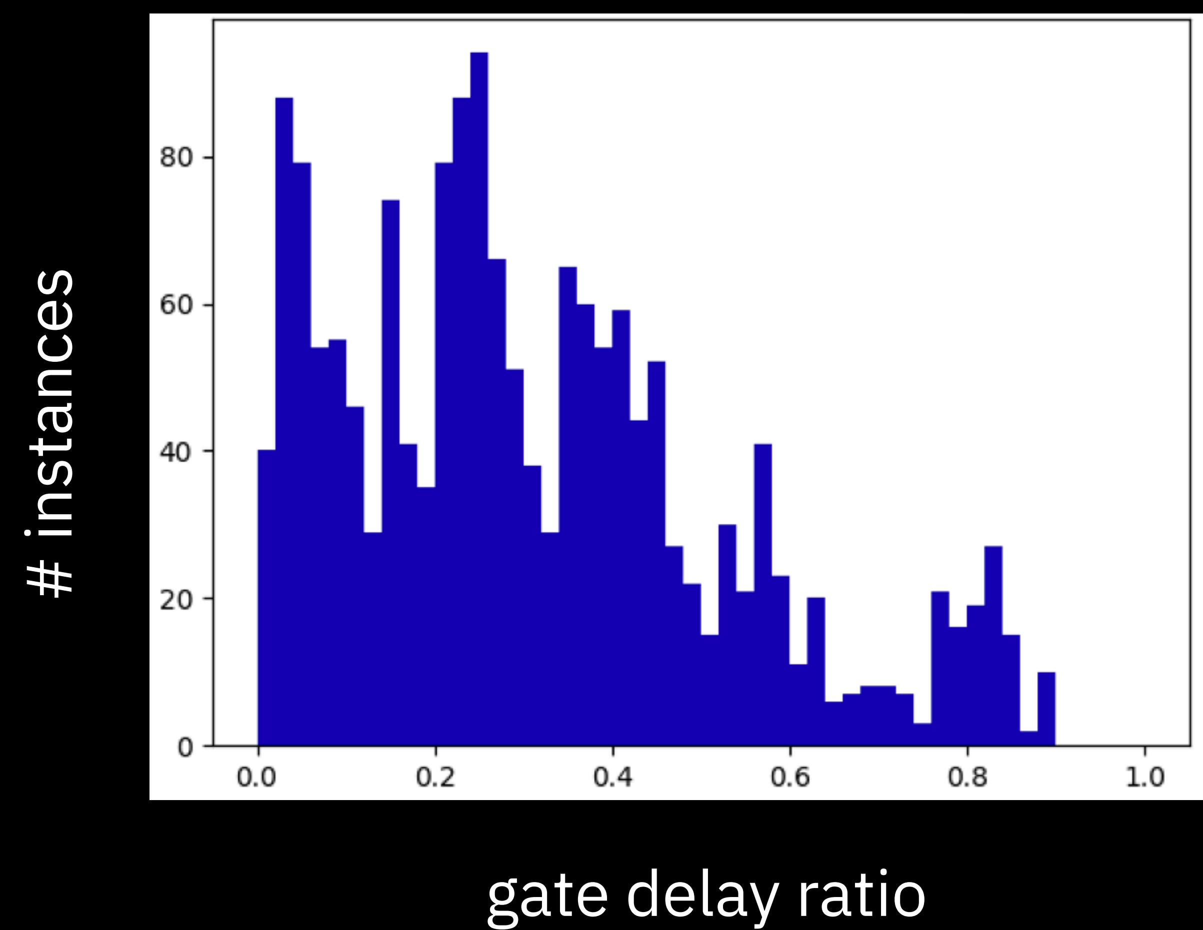


“DD has been invaluable in large scale data mining to identify systemic problems.”

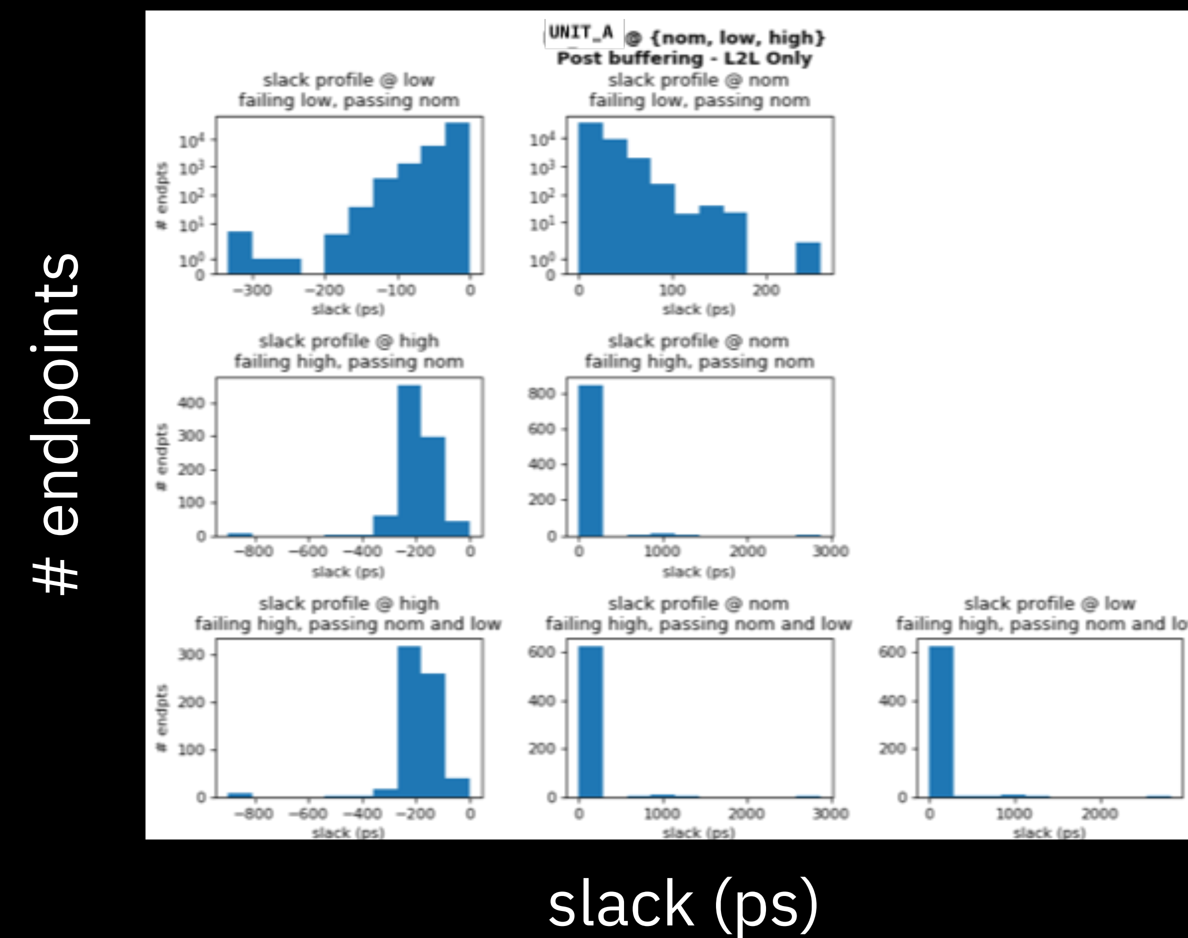
Custom analysis with pandas DataFrames and matplotlib

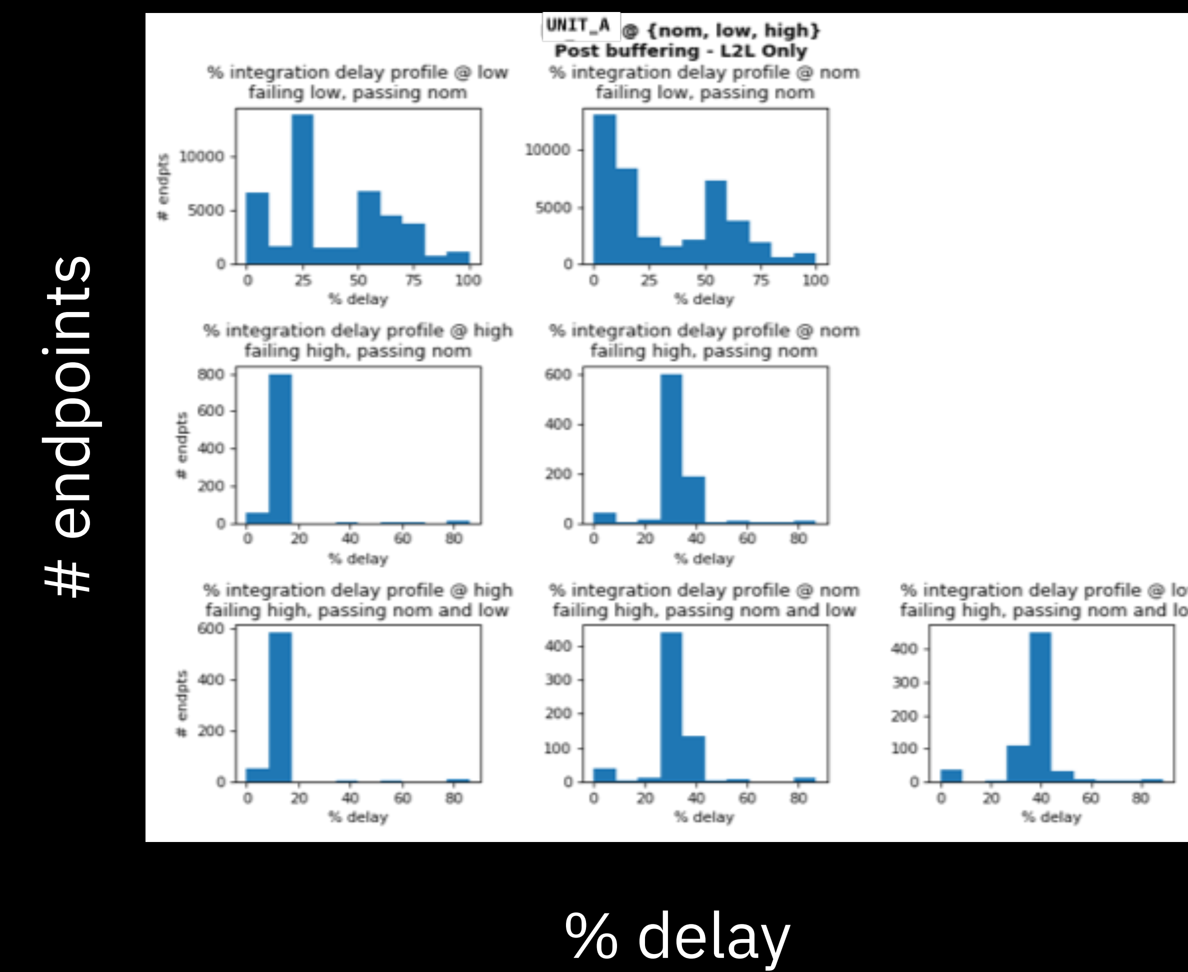
Ad-hoc analysis of **gate usage and delays**.



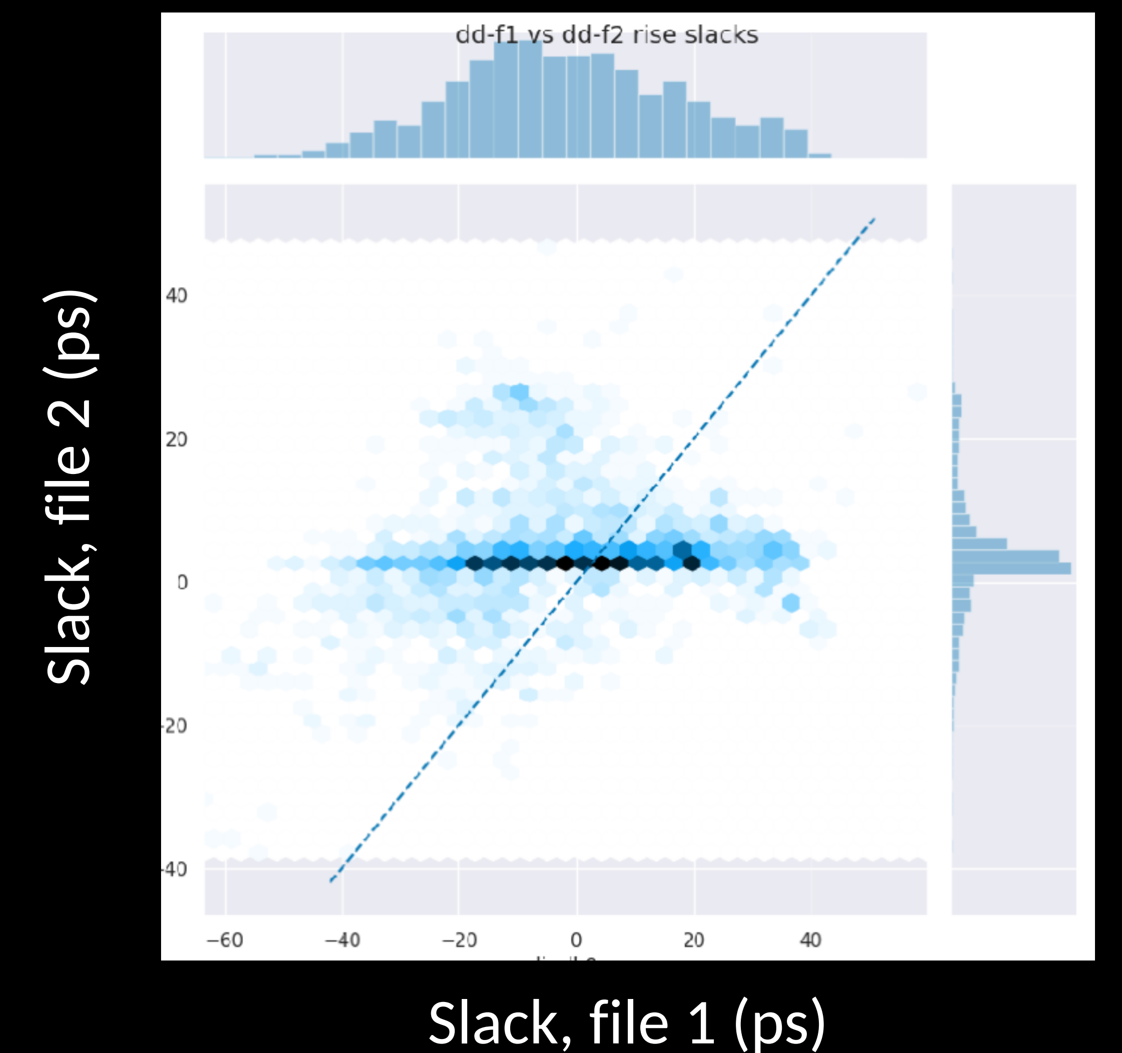


Analyze **wire delays** across multiple timing corners



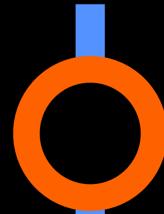


Compare **hierarchy boundary** pins between two versions

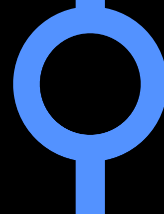


"DD has been invaluable in large scale data mining to identify systemic problems."

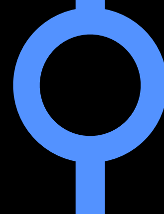
Automated regression testing



Automated regressions report many unexpected differences and significant performance degradation.



Revert PR for 44.4
Original bug remains



Commit proper fix

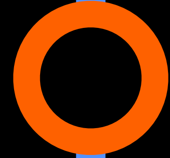
Result of Jenkins build #362

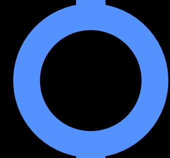
Src GraphID	Dst GraphID	CellName	Delta Fails	Macro FOM	Macro Fails	Detailed Diffs
44064 (44.3)	47009 (44.4)	TEST_A	Passed	Passed	Passed	Diffs
44065 (44.3)	47010 (44.4)	TEST_B	Passed	Passed	Passed	Passed
44066 (44.3)	47011 (44.4)	TEST_C	Failed	Failed	Failed	Failed
46520 (44.3)	47439 (44.4)	TEST_D	Passed	Passed	Passed	Diffs

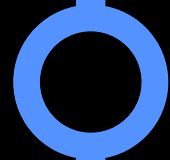
Performance for 46520 (44.3) vs 47439 (44.4)

	46520	47439	Delta
DD Read	21m 24s 908ms	25m 9s 977ms	3m 45s
Iterate Edges	4m 12s 465ms	6m 46s 682ms	2m 34s
Get Endpts	5m 45s 366ms	7m 51s 426ms	2m 6s
Analyze Timing Paths	**6m 36s 729ms**	**11m 11s 741ms**	**4m 35s**
Memory	73.859 GB	73.767 GB	

Debugging CPython applications


44.4 *Automated regressions report many unexpected differences and significant performance degradation.*


44.6 *Revert PR for 44.4 Original bug remains*


44.11 *Commit proper fix*

Where is this runtime coming from???

Experiment: Attach gdb debugger to running Python process in compute cluster.

```
> gdb -p <pid>
(gdb) bt 3     # print backtrace
#0  0x00003fffa2a523a0 in
levelize_tpts_forward(...)
   from .../site-packages/designdata.cpython-38
powerpc64le-linux-gnu.so
(More stack frames follow...)
```

Lesson Learned:
AVOID MANY CALLS TO TIME CONSUMING FUNCTIONS!

Debugging CPython applications

44.4 — *Automated regressions report many unexpected differences and significant performance degradation.*

44.6 — *Revert PR for 44.4 Original bug remains*

44.11 — *Commit proper fix*

Where is this runtime coming from???

Measure performance with Python Timer (requires successful completion)

Function Decorator

```
@Timer
def get_path_details(...):
"""aggregate path data"""
```

Context Manager

```
with Timer.getTimer('exp_a'):
    p = get_path_details(...)
```

```
(I): Performance Metrics:
(T):   Read DD file took 46m 24s
(T):   Build latch graph took 1m 11s
(T):   Iterate edges and assign groups took 5m 13s
(T):   Collect group summary took 10m 0s
(T):   timing_info.get_path_details took 4m 30s
       (0h 0m 0.006s avg) with 42126 calls.
```

Lesson learned:
AVOID MANY CALLS TO TIME CONSUMING FUNCTIONS!

pandas DataFrame “Sparse Diff”

44.4 — *Automated regressions report many unexpected differences and significant performance degradation.*

44.6 — *Revert PR for 44.4 Original bug remains*

44.11 — *Commit proper fix*

```
def sparse_diff(dfa:DataFrame, dfb:DataFrame, cols_to_compare:list, PRIMARY_KEYS:list):
    cols_with_diffs = list()
    mdf = pandas.merge(dfa, dfb, how='outer', on=PRIMARY_KEYS)
    both_df = mdf[mdf['_merge']=='both']
    for c in cols_to_compare:
        if is_number_type(both_df[c+'_a'].dtype, both_df[c+'_b'].dtype):
            both_df[c+"_d"] = both_df[c+'_a'] - both_df[c+'_b']
        else:
            both_df[c+"_d"] = (both_df[c+'_a'] != both_df[c+'_b'])
                              .replace({True: "Diff", False: "Equal"})
        both_df.loc[((both_df[c+'_d'].abs() > 1e-6) | (both_df[c+'_d'] == 'Equal')),
                    both_df.filter(regex='^'+c+'_').columns] = '-'
    return (cols_with_diffs, both_df)
```

44.3 vs. 44.4 (272 diffs)

	startPointName	Defs_41.5	Defs_44.4	bdly_d	n_gates_d	totalAdjust_d
0	XL3Q@LATC_4/QN	-	-	-	-6.0	-0.37
1	_XLQ@LATC_3/QN	PI	W_INVESLAT_X8M_A9TX	17.68	7.0	-0.21
2	XL2Q@LATC_4/QN	-	-	-	-6.0	-0.21
3	XL2Q@LATC_1/QN	-	-	-	-6.0	-
4	INST@LATC_8/QN	-	-	-	-6.0	-
5	INST@LATC_6/QN	INVESLAT_X1M_A9TX	INVESLATN_X1M_A9TS	-13.85	-8.0	-24.54

44.3 vs. 44.11 (2 diffs)

	startPointName	Defs_41.5	Defs_44.11	bdly_d	n_gates_d	totalAdjust_d
0	_XLQ@LATC_3/QN	PI	W_INVESLAT_X8M_A9TX	17.68	7.0	-0.21
1	INST@LATC_6/QN	INVESLAT_X1M_A9TX	_INVESLATN_X1M_A9TS	-13.85	-8.0	-24.54

Data Compare Performance

Use a vectorized approach

1. Iterate over columns and compare all rows per-column **"vector-wise"**

```
for c in dfa.columns:
    diffs[c] = dfb[c] - dfa[c]
```

2. DataFrame.compare()

```
diffs = dfa.compare(dfb)
```

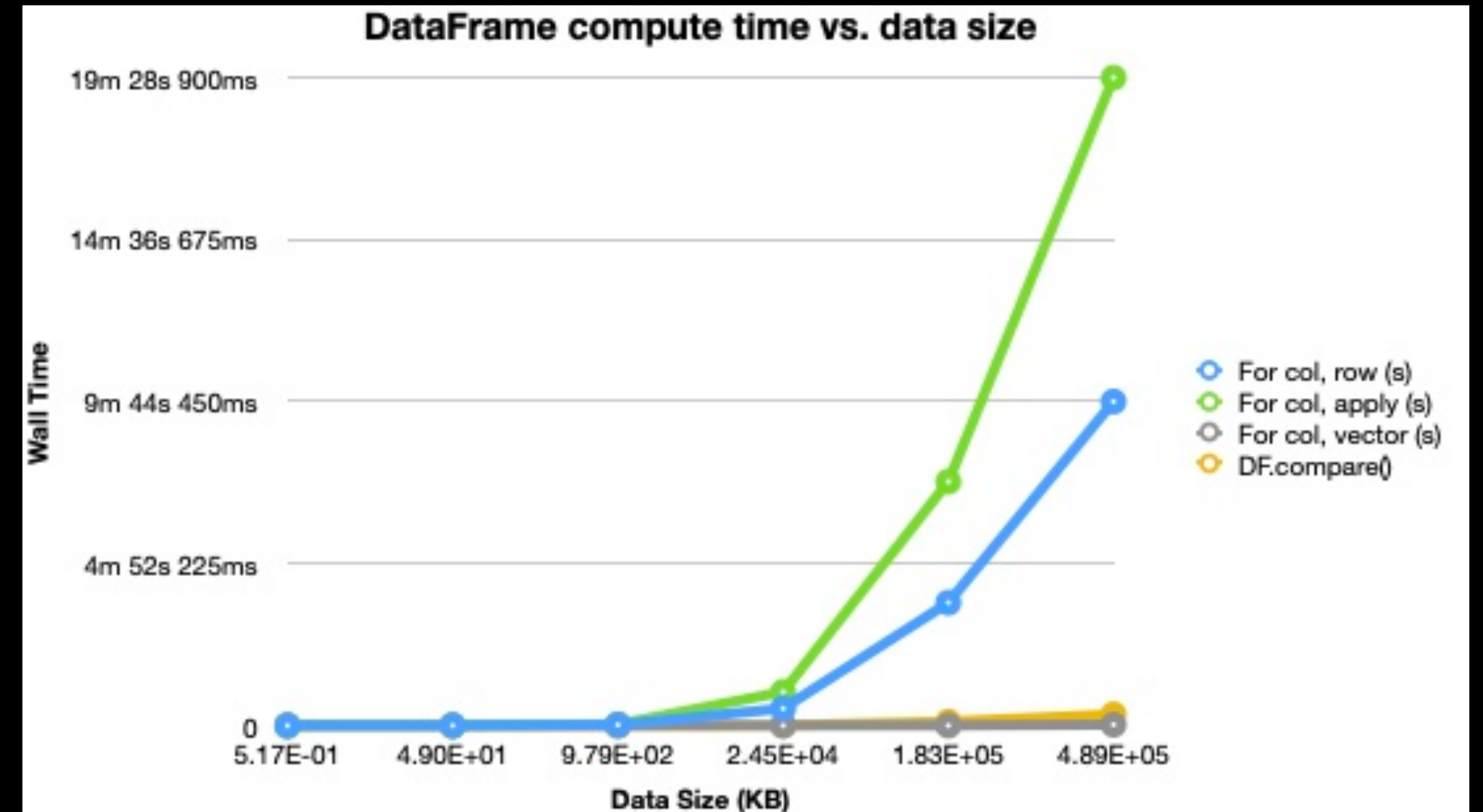
3. Iterate over columns and rows to compare **element-wise**

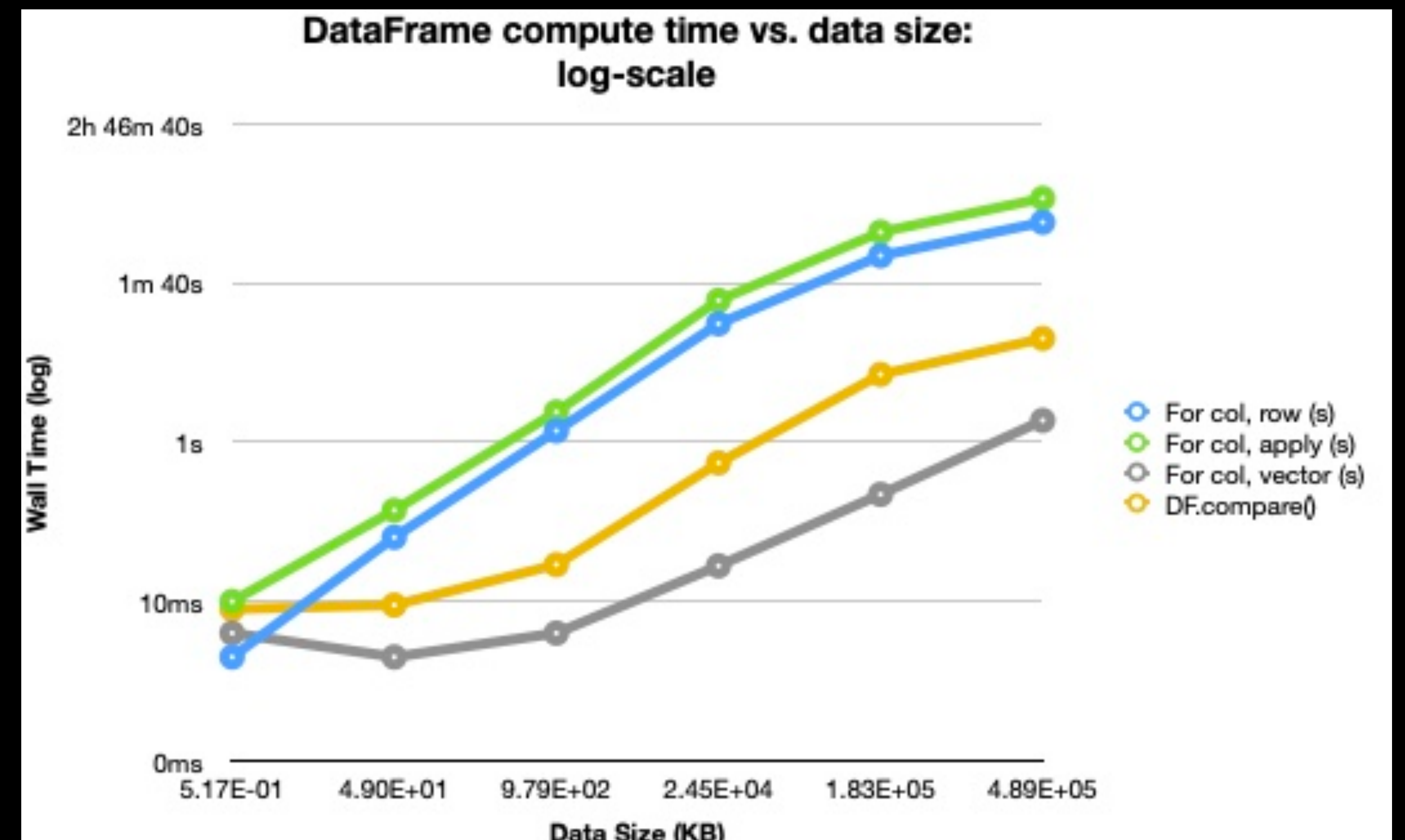
```
for c in dfa.columns:
    for i, ri_a in enumerate(dfa[c]):
        diffs[c].append(dfb[c][i] - ri_a)
```

4. Iterate over columns and use `DataFrame.apply()` to compare **element-wise**:

```
for c in cols:
    diffs[c].append(dfM.apply(
        lambda row: row[c+'_a'] - row[c+'_b'], axis=1))
```

*The "vectorized" approach had more than **500x improvement** over the other loop-based methods.*





Objectives

I.

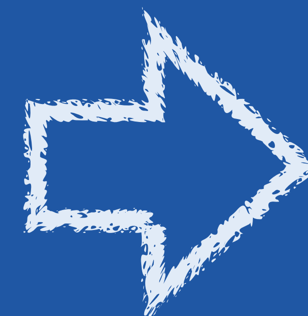
How to build a microprocessor: **a Big Data Problem**

II.

An efficient **CPython** data model

III.

Using **Python** for data analysis

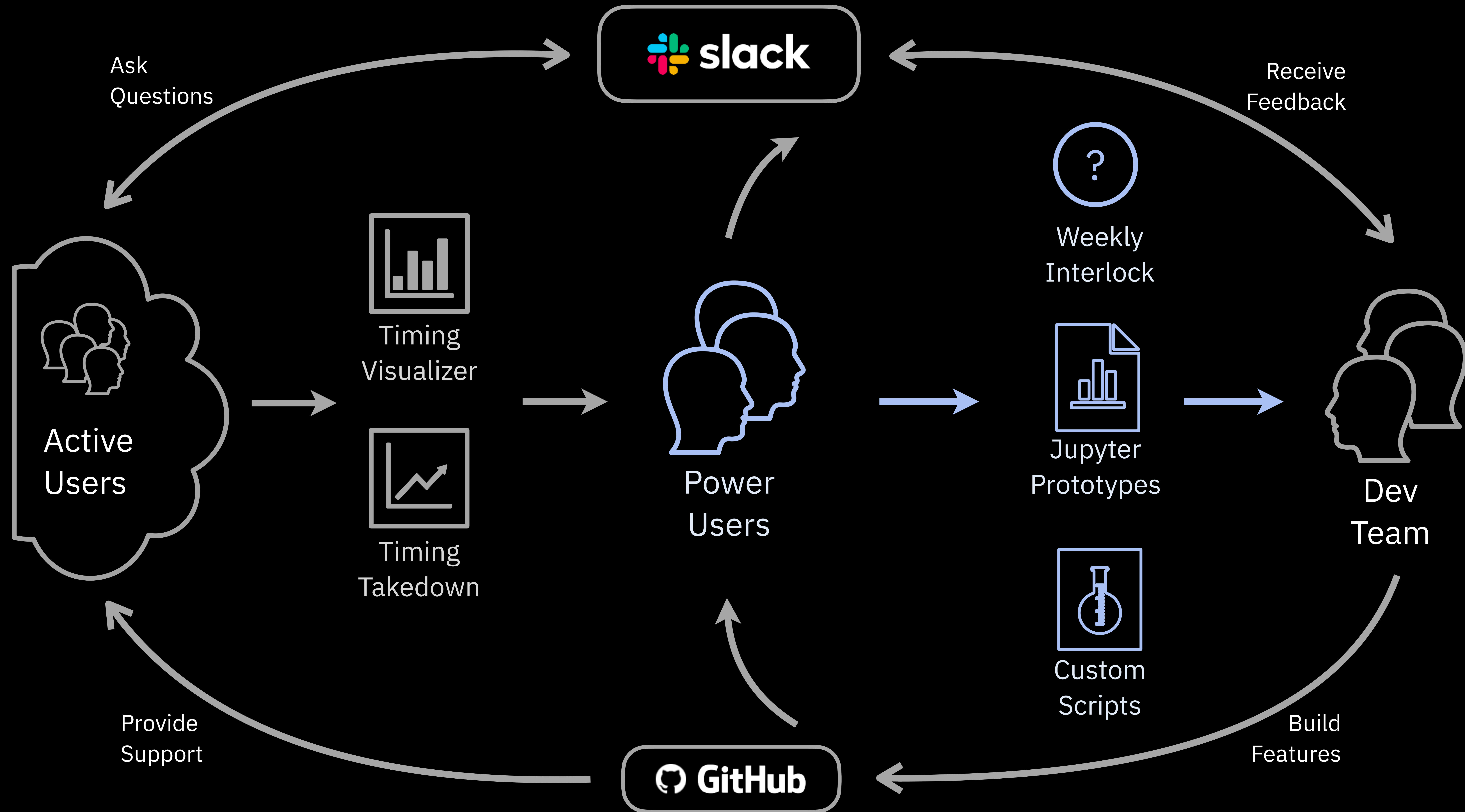


IV.

Lessons learned & Wrap-up

Open Source Community Model

Democratized Data Analysis



"DD makes it practical for ordinary engineers to perform their own analysis without specialized EDA help!"

I want this!

How do I get it?

Make one!

Here are some references to help you get started.

Learn Python!

Learn C or C++!

Basic Concepts, Syntax, Grammar

Learn CPython!

Create a C / C++ Extension Module

Python Standard Library
https://docs.python.org/3/library/index.html

The Python Tutorial
https://docs.python.org/3/tutorial/index.html

C and C++ Standard Library
https://en.cppreference.com/w/

C++ Tutorial
https://www.cplusplus.com/doc/tutorial/

CPython: Defining Extension Types
https://docs.python.org/3/extending/newtypes_tutorial.html

What have we learned?

DD IS A GAME CHANGER!

- Significant reduction in memory footprint
- Enables **data-driven** design using a complete data model
- A **Python** interface allows engineers to apply existing methods from **Data Science** and focus on the **hard problems**!