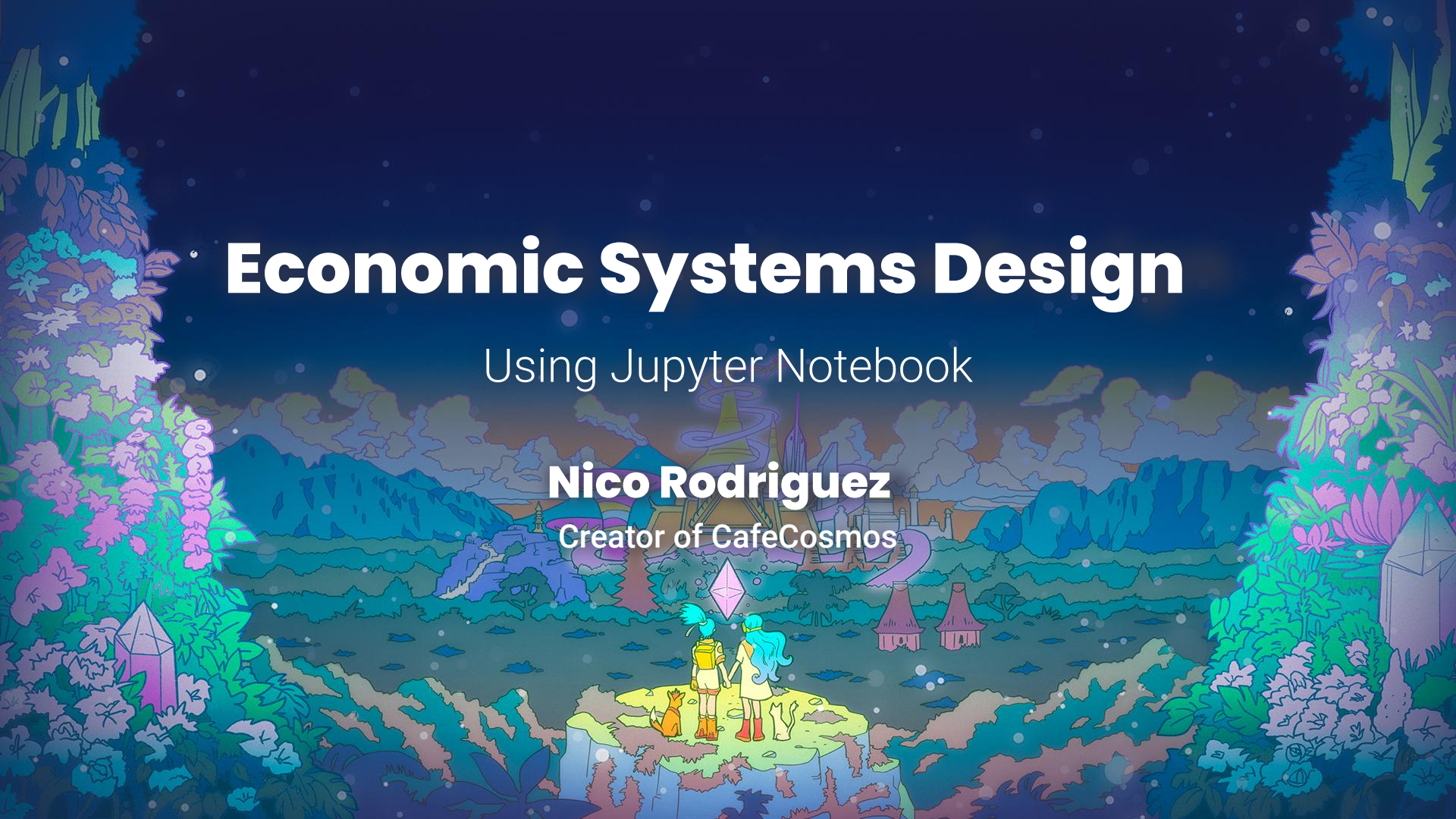
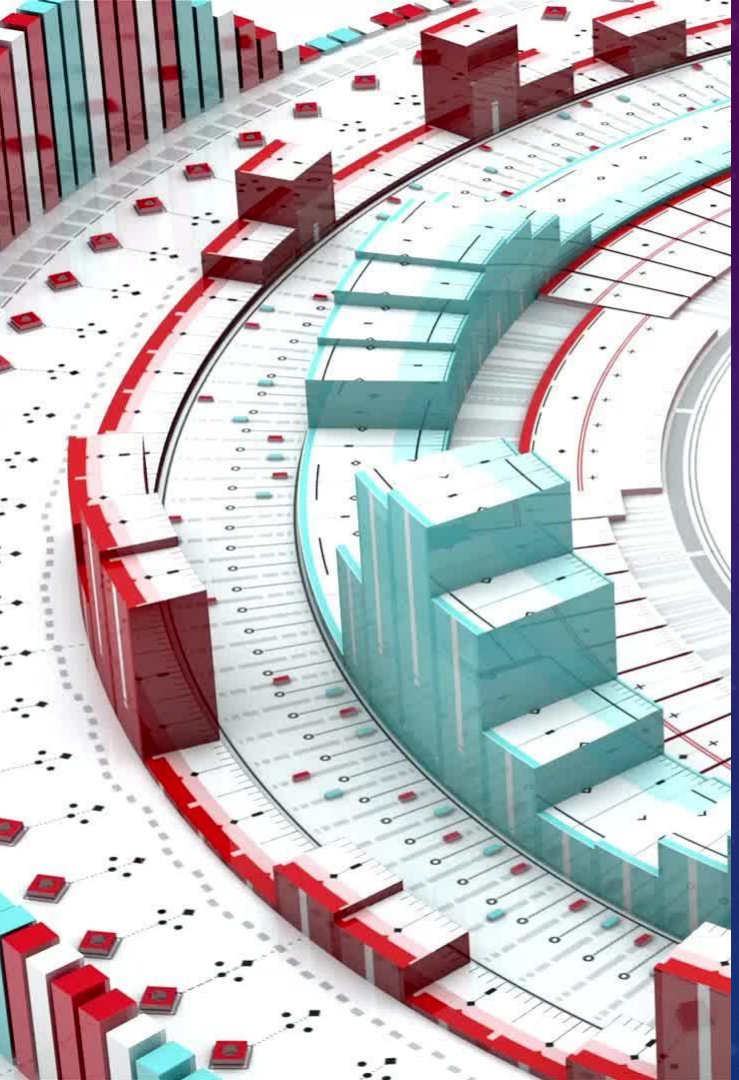

Economic Systems Design
Using Jupyter Notebook
Nico Rodriguez
Creator of CafeCosmos



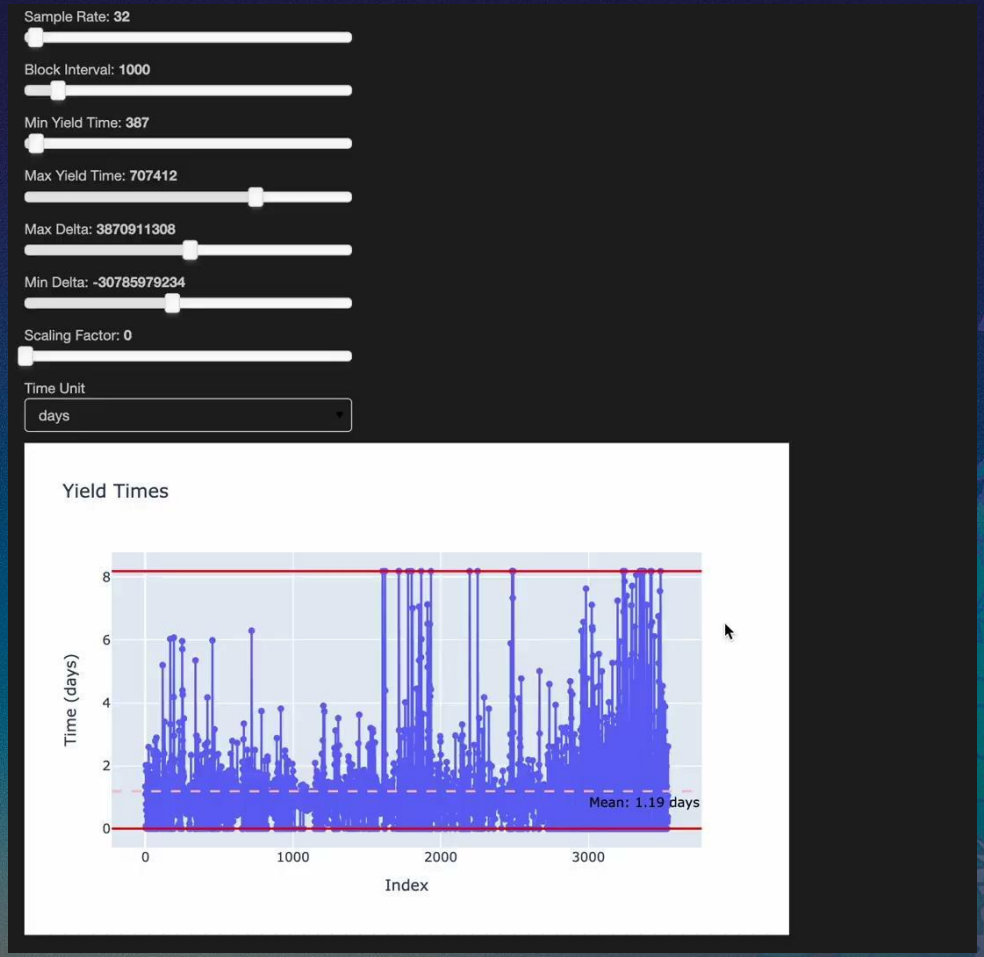
COMPLEX SYSTEMS IN AW'S

- Autonomous World's functionality as a collection of systems
- How do we build increasingly complex systems

Crafting, earning, exploring, trading, fighting, and more

Every CafeCosmos system in 43 seconds

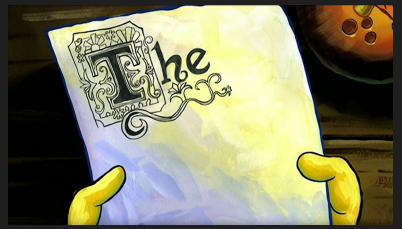
How to make an Economic simulation for your complex system



TRYING TO DESIGN SYSTEMS WHILE LOOKING AT A DARK IDE

really_complicated.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
```

INTRO: DIGITAL TWINS

"A digital copy of your system in a form you can easily simulate over and interact with"…

…In this case, your system as a python class

The simulation will be how we manually or automatically interact with that system to acquire visual results

INTRO: DIGITAL TWINS

State: The “mock” blockchain state

Onchain Functions: The functions on your smart contract

Display Function: How you visualize your results for your understanding

Simulation: How you quickly play with and achieve results from your system

State

Onchain Functions

Display Function

Simulation

```python
CHUNK_SIZE = 15


class Land:

    def __init__(self, lim_x=0, lim_y=0):
        self.lim_x = lim_x
        self.lim_y = lim_y
        self.y_bound = []
        self.plot = [[0] * lim_x for _ in range(lim_y)]

    def add_chunk(self):
        count = 0
        for y in range(len(self.y_bound)):
            for x in range(self.y_bound[y], self.lim_x):
                self.initialize_slot(x, y)
                self.y_bound[y] += 1
                count += 1
                if(count == CHUNK_SIZE):
                    break
            if(count == CHUNK_SIZE):
                break



    def initialize_slot(self, x, y):
        self.plot[y][x] = 1

    def expand_land(self, x2, y2):
        for i in range(y2):
            self.y_bound.append(0)

        # Expand each existing row by x2
        for row in self.plot:
            row.extend([0] * x2)

        # Add new rows increased by y2
        new_rows = [0] * (self.lim_x + x2)
        for _ in range(y2):
            self.plot.append(new_rows[:])  # Create a copy of
new_rows
        # Update the limits
        self.lim_x += x2
        self.lim_y += y2

    def check_bounds(self, x, y):
        return y < self.lim_y and x < self.y_bound[y]

    def display_land(self):
        for row in reversed(self.plot):
            for element in row:
                if element == 0:
                    print("⬛", end=' ')
                elif element == 1:
                    print("🟦", end=' ')
                else:
                    print("? ", end=' ')
            print()

land = Land()
land.expand_land(10, 10)
```

WHY DO THIS:

Reduce perceived complexity while doing systems design

Give yourself a mental map of what you're doing

Visualization helps design

Faster iteration

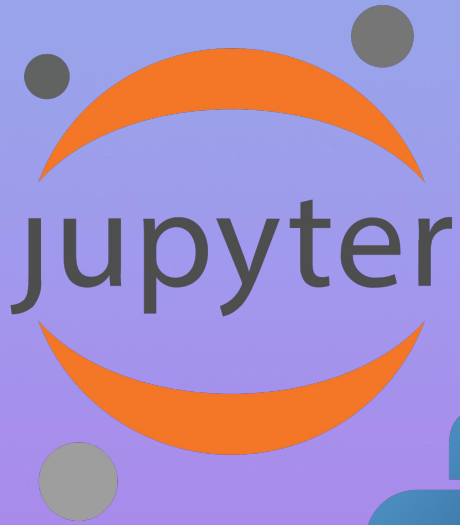
WHY PYTHON:

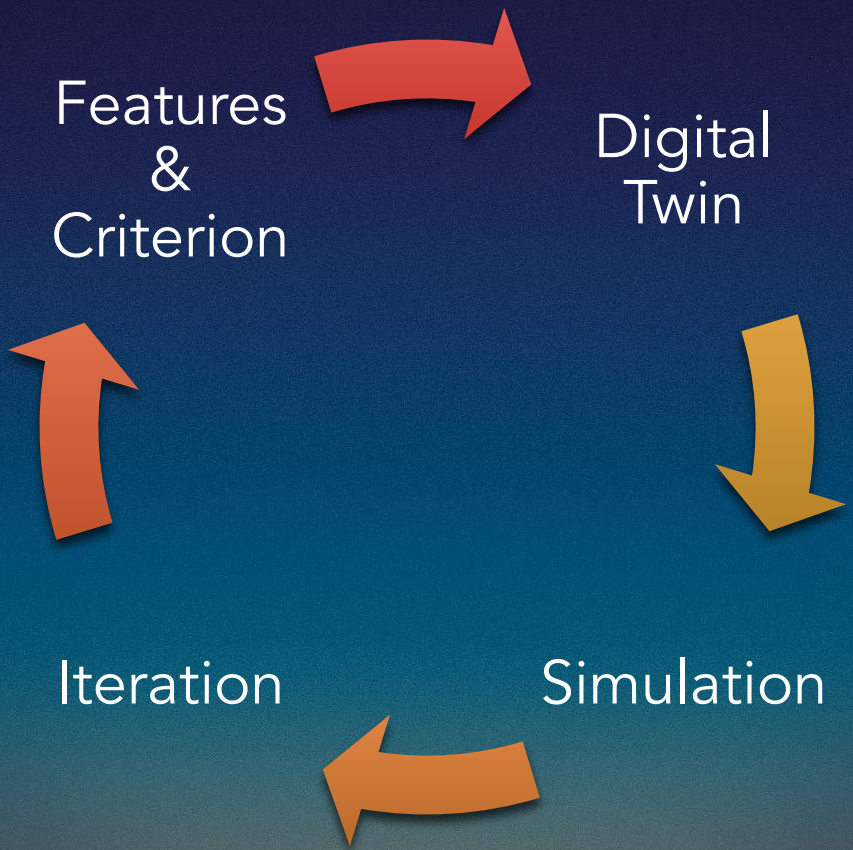
Endless [interactive] data visualization libraries

Easy to read for others

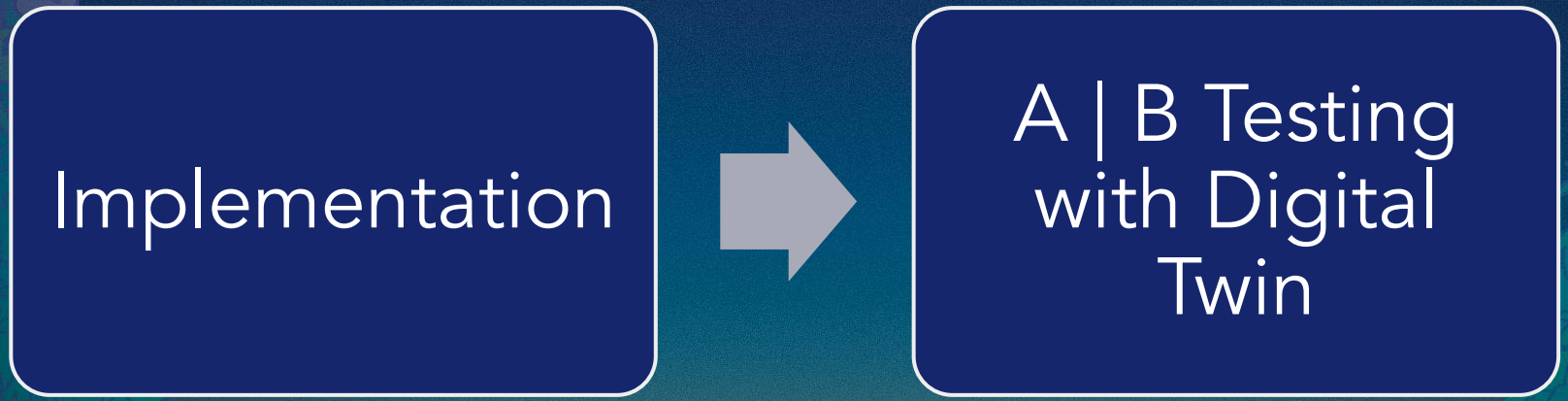
Jupyter notebook allows for trains of thought through text and code

Easy to transcribe algorithms




Features
&
Criterion
Digital
Twin
Simulation
Iteration

+ INTEGRATION AND A | B TESTING



DESIGNING YOUR SYSTEM: FEATURES AND CONSTRAINTS

	Feature Combination A	Feature Combination B	…
Constraint 1			
Constraint 2			
…			

PROBLEM 1: PERLIN LAND GENERATION (MAP INITIALIZATION)

Instantiation 🐣

PROBLEM 1: PERLIN LAND GENERATION

GENERATE ALL AT ONCE

	Load all at once
gas doesn’t run out	
simple algorithm	✅
Looks good	✅

PROBLEM 1: PERLIN LAND GENERATION

GENERATE ALL AT ONCE

```
function load_land(uint256 x_lim, uint256 y_lim, uint256 seed) internal
{
    for(uint256 y = 0; y < y_lim; y++){
        for(uint256 x = 0; x < x_lim; x++){
            uint256 itemInSlot = calculatePerlinAndGetItem(x, y, seed);
            landSlot.set(x, y, itemInSlot);
        }
    }
}
```

```
[1056943993] CounterTest::test_Increment()
  ├─ [1056936571] Counter::increment()
  │   └─ ← [OutOfGas] EvmError: OutOfGas
  └─ ← [Revert] EvmError: Revert
```

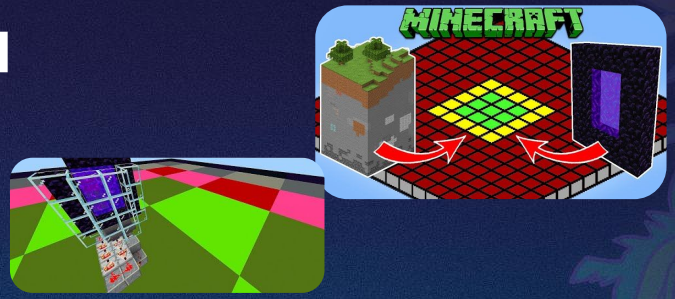
PROBLEM 1: PERLIN LAND GENERATION

GENERATE ALL AT ONCE

	Load all at once
gas doesn't run out	❌
simple algorithm	✅
Looks good	✅

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + SQUARE CHUNK LOADING



	Load all at once	Static size + square chunk loading
gas doesn't run out	❌	~✅
simple algorithm	✅	
Looks good	✅	✅

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + SQUARE CHUNK LOADING

MOCKING UP WITH JUPYTER NOTEBOOK

	Load all at once	Static size + square chunk loading
gas doesn't run out	❌	~✅
simple algorithm	✅	
Looks good	✅	✅

YOU HAVE TO UNROLL THE SQUARE

TOO COMPLICATED?

YUP

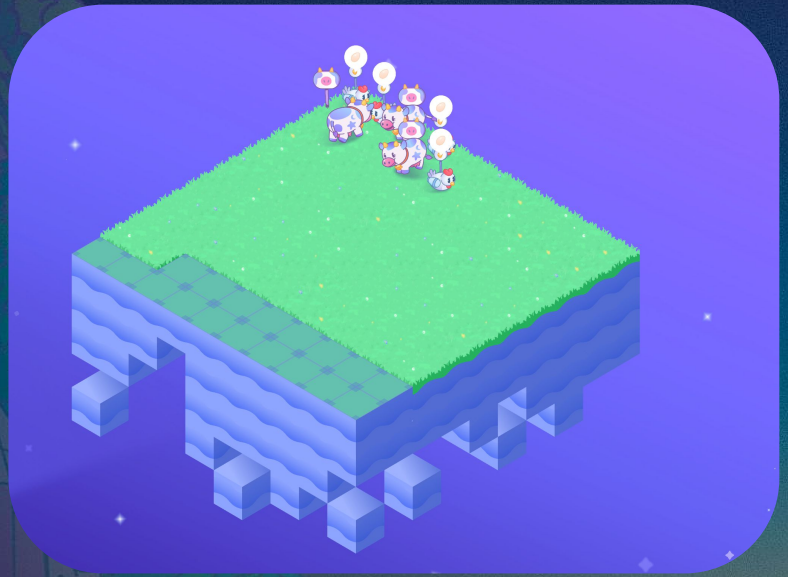
	Load all at once	Static size + square chunk loading
gas doesn't run out	❌	~✅
simple algorithm	✅	❌
Looks good	✅	✅

```python
def place_chunk(self):
    if self.is_full():
        return False

    # Try to find next available position
    row, col = self.next_position
    placed = False

    while row < self.length and not placed:
        while col < self.width and not placed:
            if self.can_place_chunk(row, col):
                # Fill the chunk area with current_fill value
                for i in range(row, row + self.chunk_size):
                    for j in range(col, col + self.chunk_size):
                        self.grid[i][j] = self.current_fill

                placed = True
                # Update next position to try
                self.next_position = (row, col + self.chunk_size)
                if col + self.chunk_size >= self.width:
                    self.next_position = (row + self.chunk_size,
                self.current_fill += 1
0)
            else:
                col += 1
        if not placed:
            row += 1
            col = 0

    return placed
```

```
After placing chunk:
 1  1  0  0  0  0
 1  1  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0

After placing chunk:
 1  1  2  2  0  0
 1  1  2  2  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0

After placing chunk:
 1  1  2  2  3  3
 1  1  2  2  3  3
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0

After placing chunk:
 1  1  2  2  3  3
 1  1  2  2  3  3
 4  4  0  0  0  0
 4  4  0  0  0  0
 0  0  0  0  0  0

After placing chunk:
 1  1  2  2  3  3
 1  1  2  2  3  3
 4  4  5  5  0  0
 4  4  5  5  0  0
 0  0  0  0  0  0

After placing chunk:
 1  1  2  2  3  3
 1  1  2  2  3  3
 4  4  5  5  6  6
 4  4  5  5  6  6
 0  0  0  0  0  0

Cannot place more chunks!
```

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + LINEAR CHUNK LOADING



	Load all at once	Static size + square chunk loading	Static size + Linear Loading
gas doesn't run out	❌	~✅	
simple algorithm	✅	❌	
Looks good	✅	✅	~✅

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + LINEAR CHUNK LOADING

Linear

```
    def add_chunk(self):
        count = 0
        for y in range(len(self.y_bound)):
            for x in range(self.y_bound[y],
self.lim_x):    self.initialize_slot(x, y)
                self.y_bound[y] += 1
                count += 1
                if(count == CHUNK_SIZE):
                    break
            if(count == CHUNK_SIZE):
                break

```

Square

```
    def place_chunk(self):
        if self.is_full():
            return False

        # Try to find next available position
        row, col = self.next_position
        placed = False

        while row < self.length and not placed:
            while col < self.width and not placed:
                if self.can_place_chunk(row, col):
                    # Fill the chunk area with current_fill value
                    for i in range(row, row + self.chunk_size):
                        for j in range(col, col + self.chunk_size):
                            self.grid[i][j] = self.current_fill

                    placed = True
                    # Update next position to try
                    self.next_position = (row, col + self.chunk_size)
                    if col + self.chunk_size >= self.width:
                        self.next_position = (row + self.chunk_size,
0)
                    self.current_fill += 1
                else:
                    col += 1
            if not placed:
                row += 1
                col = 0

        return placed
```

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + LINEAR CHUNK LOADING

```
    def add_chunk(self):
        count = 0
        for y in range(len(self.y_bound)):
            for x in range(self.y_bound[y],
self.lim_x):    self.initialize_slot(x, y)
                self.y_bound[y] += 1
                count += 1
                if(count == CHUNK_SIZE):
                    break
            if(count == CHUNK_SIZE):
                break

```

	Load all at once	Static size + square chunk loading	Static size + Linear Loading
gas doesn't run out	❌	~✅	~✅
simple algorithm	✅	❌	✅
Looks good	✅	✅	~✅

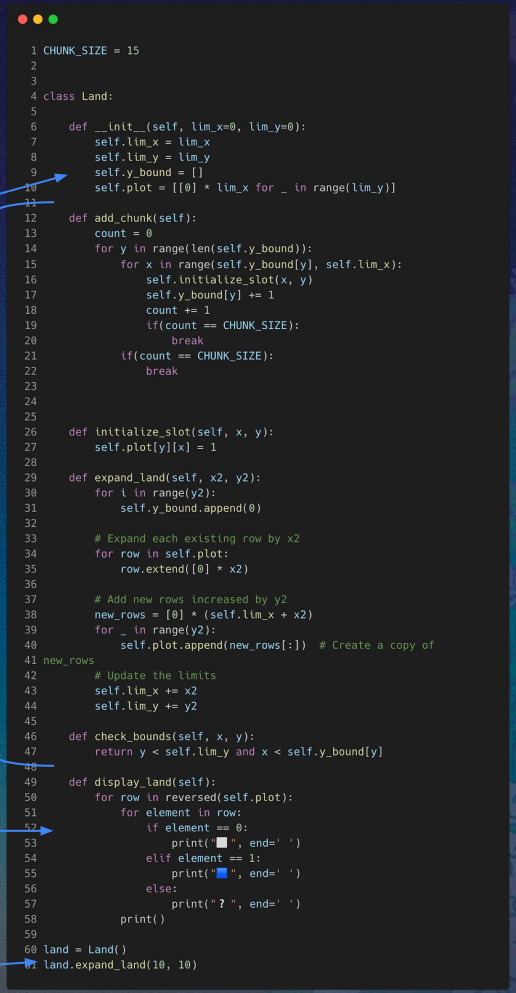
MAKING A DIGITAL TWIN! 😱

State

Onchain Functions

Display Function

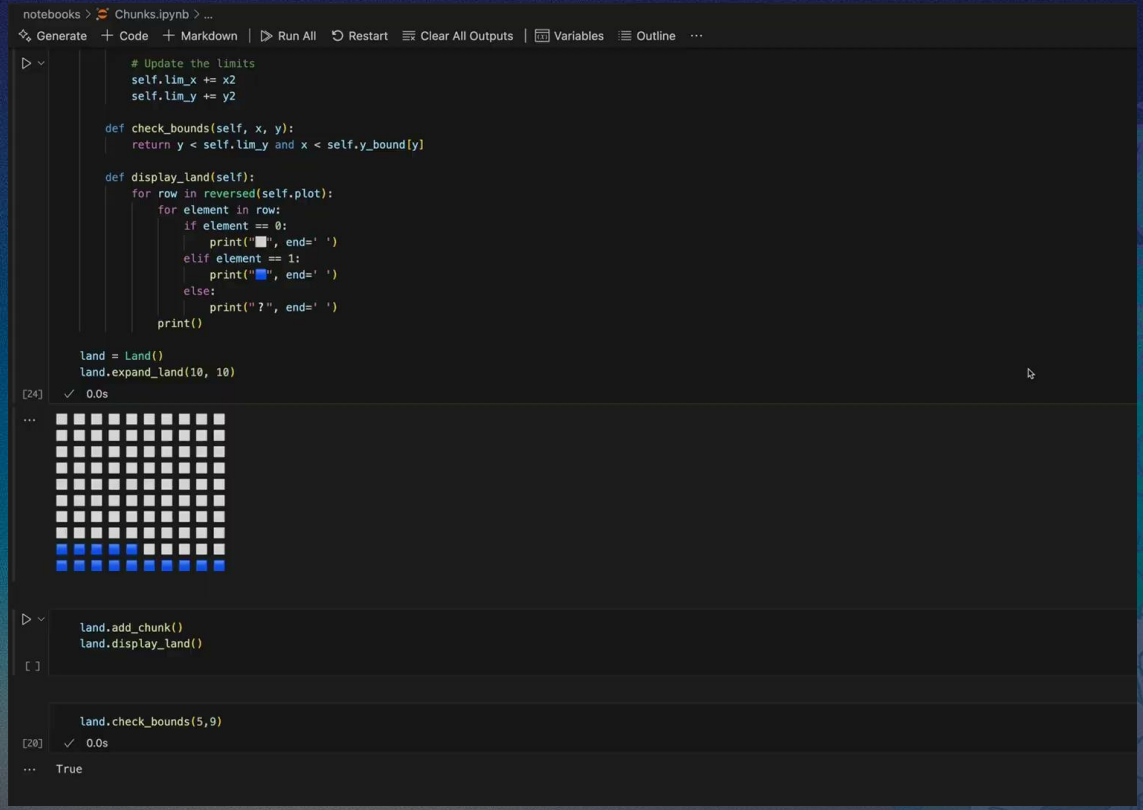
Initialization



```python
CHUNK_SIZE = 15


class Land:

    def __init__(self, lim_x=0, lim_y=0):
        self.lim_x = lim_x
        self.lim_y = lim_y
        self.y_bound = []
        self.plot = [[0] * lim_x for _ in range(lim_y)]

    def add_chunk(self):
        count = 0
        for y in range(len(self.y_bound)):
            for x in range(self.y_bound[y], self.lim_x):
                self.initialize_slot(x, y)
                self.y_bound[y] += 1
                count += 1
                if(count == CHUNK_SIZE):
                    break
            if(count == CHUNK_SIZE):
                break



    def initialize_slot(self, x, y):
        self.plot[y][x] = 1

    def expand_land(self, x2, y2):
        for i in range(y2):
            self.y_bound.append(0)

        # Expand each existing row by x2
        for row in self.plot:
            row.extend([0] * x2)

        # Add new rows increased by y2
        new_rows = [0] * (self.lim_x + x2)
        for _ in range(y2):
            self.plot.append(new_rows[:])  # Create a copy of
new_rows
        # Update the limits
        self.lim_x += x2
        self.lim_y += y2

    def check_bounds(self, x, y):
        return y < self.lim_y and x < self.y_bound[y]

    def display_land(self):
        for row in reversed(self.plot):
            for element in row:
                if element == 0:
                    print("⬜", end=' ')
                elif element == 1:
                    print("🟦", end=' ')
                else:
                    print("❓", end=' ')
            print()

land = Land()
land.expand_land(10, 10)
```

PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + LINEAR CHUNK LOADING



PROBLEM 1: PERLIN LAND GENERATION

STATIC SIZE + LINEAR CHUNK LOADING

	Load all at once	Static size + square chunk loading	Static size + Linear Loading
gas doesn't run out	❌	~✅	~✅
simple algorithm	✅	❌	✅
Looks good	✅	✅	~✅

PROBLEM 1: PERLIN LAND GENERATION

DYNAMIC SIZE + LINEAR CHUNK LOADING

	Load all at once	Static size + square chunk loading	Static size + Linear Loading	Dynamic size + Linear Loading + multiple tx's from frontend
gas doesn't run out	❌	~✅	~✅	✅
simple algorithm	✅	❌	✅	✅
Looks good	✅	✅	~✅	✅

PROBLEM 1: PERLIN LAND GENERATION

DYNAMIC SIZE + LINEAR CHUNK LOADING

```
1       function generateChunk(uint256 landId) public onlyLandOperator(landId)
2 {         uint256 count = 0;
3           uint256[] memory yBounds = LandInfo.getYBound(landId);
4           uint256 limitX = LandInfo.getLimitX(landId);
5
6           for (uint256 y = 0; y < yBounds.length; y++) {
7               uint256 ybound = yBounds[y];
8               for (uint256 x = ybound; x < limitX; x++) {
9                   ybound = ybound + 1;
10                  initialiseItem(landId, x, y);
11                  LandInfo.updateYBound(landId, y, ybound);
12                  count++;
13                      break;
14                  }
15              if (gasleft() < 10000) {
16                  break;
17              }
18          }
19      }
```

A | B TESTING

- Acquire same state in digital twin and implementation
- Create assertions to validate same results when running same functions

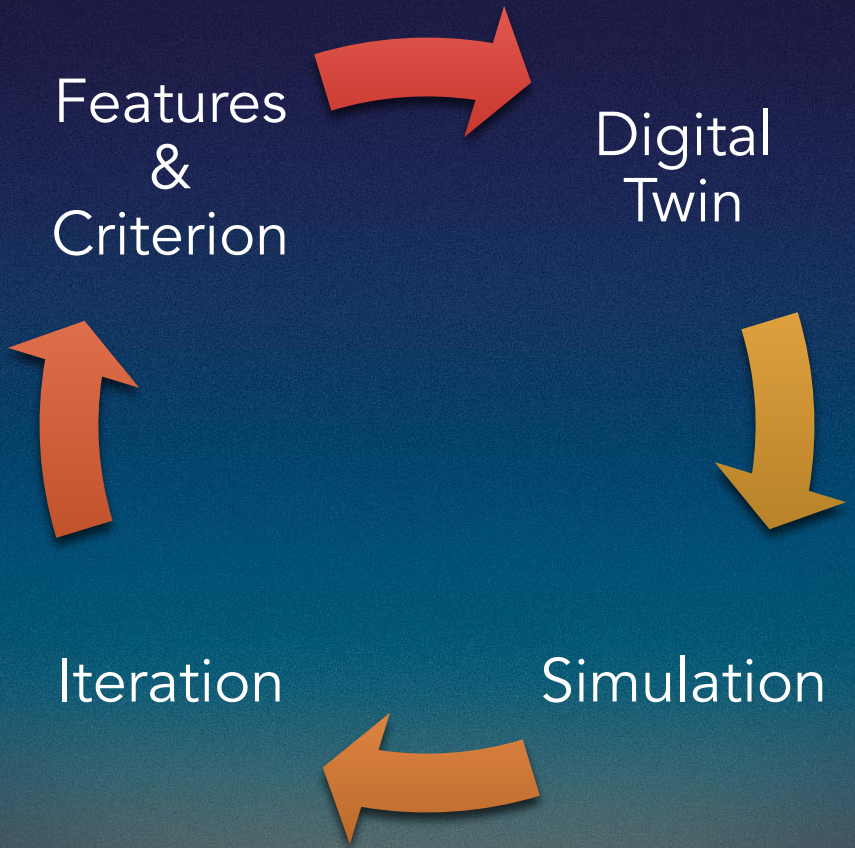
Solidity

```
1 land.generateChunk(landId);
2 Assert(land.checkBounds(0,7))
```

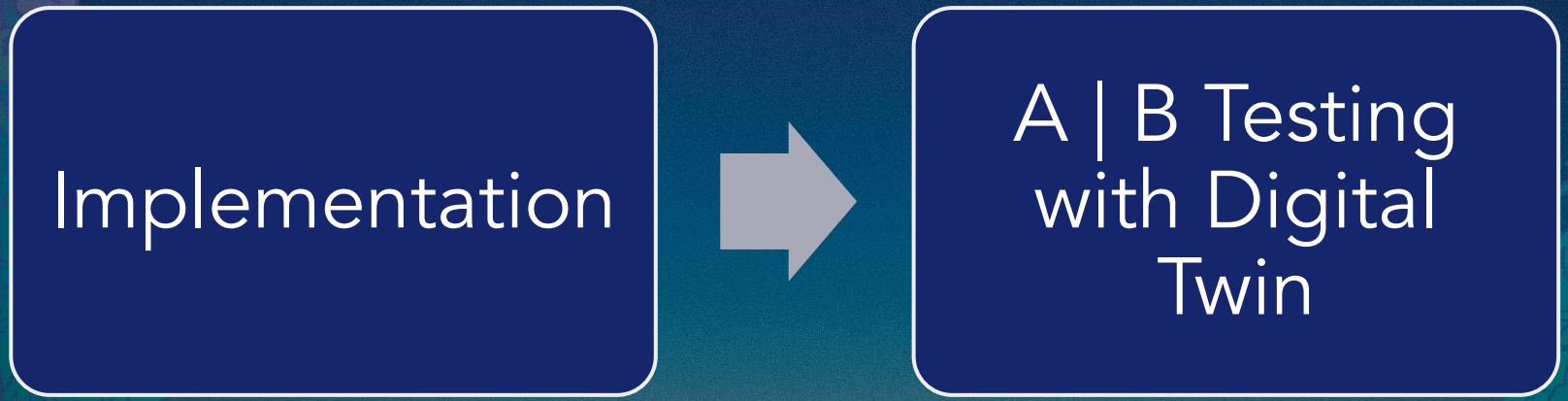
Python

```
1 land.generateChunk(landId);
2 land.checkBounds(0,7)
3
4 >> True
```

Bringing it back…



+ INTEGRATION AND A | B TESTING



PROBLEM 2: IN GAME ITEM INFLATION

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry
Bot resistant	❌
simple algorithm	✅
Looks good	✅
Feels natural	✅
Not too upsetting	✅
Doesn’t require oracle	✅
Requires Strategy	❌

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry
Bot resistant	❌
simple algorithm	✅
Looks good	✅
Feels natural	✅
Not too upsetting	✅
Doesn’t require oracle	✅
Requires Strategy	❌

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry	Food expiry + Tree disease at random
Bot resistant	❌	~✅
simple algorithm	✅	❌
Looks good	✅	✅
Feels natural	✅	✅
Not too upsetting	✅	❌
Doesn’t require oracle	✅	❌
Requires Strategy	❌	❌

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry	Food expiry + Tree disease at random	Rain allows produce to grow at random
Bot resistant	❌	~✅	~✅
simple algorithm	✅	❌	❌
Looks good	✅	✅	✅
Feels natural	✅	✅	✅
Not too upsetting	✅	❌	❌
Doesn’t require oracle	✅	❌	❌
Requires Strategy	❌	❌	~❌

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry	Food expiry + Tree disease at random	Rain allows produce to grow at random	Plants require water to grow and the time of water well extraction is based on the derivate of gas with an oracle
Bot resistant	❌	~✅	~✅	✅
simple algorithm	✅	❌	❌	❌
Looks good	✅	✅	✅	✅
Feels natural	✅	✅	✅	✅
Not too upsetting	✅	❌	❌	~✅
Doesn't require oracle	✅	❌	❌	❌
Requires Strategy	❌	❌	~❌	✅

PROBLEM 2: IN GAME ITEM INFLATION

	Food expiry	Food expiry + Tree disease at random	Rain allows produce to grow at random	Plants require water to grow and the time of water well extraction is based on the derivate of gas with an oracle	Plants require water to grow and the time of water well extraction is based on the derivate of eth with storage proofs
Bot resistant	❌	~✅	~✅	✅	✅
simple algorithm	✅	❌	❌	❌	~✅
Looks good	✅	✅	✅	✅	✅
Feels natural	✅	✅	✅	✅	✅
Not too upsetting	✅	❌	❌	~✅	~✅
Doesn't require oracle	✅	❌	❌	❌	✅
Requires Strategy	❌	❌	~❌	✅	✅

- Plants require **water** to **grow** AND the time of water well extraction is based on the **derivative** of eth proven with storage proofs

VARIABLES

- Block Intervals for TWAPs

Base fees (60s) = [$5.69, $4.75, $4.49, $4.51, $4.56, $4.61, $7.69]

VARIABLES

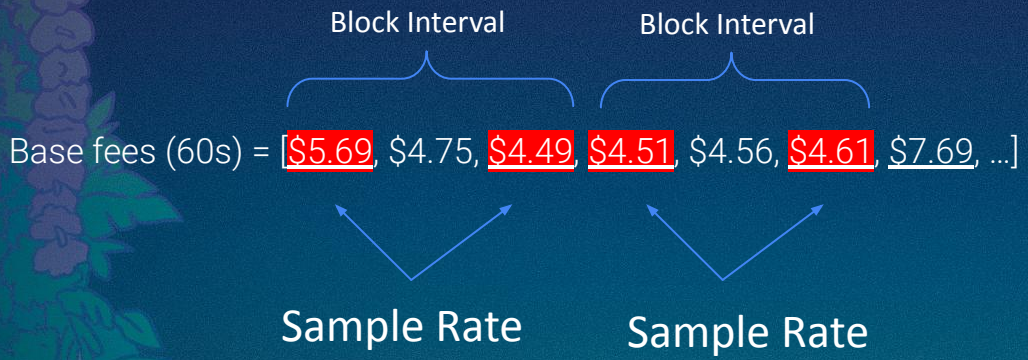
- Block Intervals for TWAPs

Block Interval = 3

Base fees (60s) = [$5.69, $4.75, $4.49, $4.51, $4.56, $4.61, $7.69, ...]

VARIABLES

- Block Intervals for TWAP's
- Sample Rate for TWAP's



Block Interval = 3
Sample Rate = 2

VARIABLES

- Block Intervals for TWAP’s
- Sample Rate for TWAP’s
- Min Yield Time
- Max Yield Time

How long can we bear at the slowest,
How little can we wait at the fastest

VARIABLES

- Block Intervals for TWAP’s
- Sample Rate for TWAP’s
- Min Yield Time
- Max Yield Time
- Min Delta
- Max Delta

How much change are we willing to tolerate at highest and at lowest?

SET VARIABLES

```
NUM_SAMPLES = 32
BLOCK_INTERVAL = 1000
MIN_YIELD_TIME = 387
MAX_YIELD_TIME = 707412
MIN_DELTA = -54067095249
MAX_DELTA = 3870911308
OBSERVATION_PERIOD = BLOCK_INTERVAL*2*365
```

STEP 0: LET'S GET SOME DATA!

STEP 0: LET'S GET SOME DATA!

- Use Alchemy's API to download every basefee ever!

.. YES IT'S THERE

```python
import csv

# Define the file path
file_path = './data/dl_fees.csv'

# Write the dl_fees_int array to the CSV file
with open(file_path, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(dl_fees_int)

```

```python
def get_fees(blocks: int, latest_block: str):
    assert blocks < 1025
    payload = {
        "id": 1,
        "jsonrpc": "2.0",
        "method": "eth_feeHistory",
        "params": [blocks, latest_block]
    }
    headers = {
        "accept": "application/json",
        "content-type": "application/json"
    }

    response = requests.post(url, json=payload, headers=headers)

    print(response.text)
    data = response.json()

    return (data['result']['oldestBlock'], data['result']["baseFeePerGas"])

def download_fees():
    fees = []
    cur_top = LATEST_BLOCK
    for i in range(OBSERVATION_PERIOD//1024):
        res = get_fees(1024, cur_top)
        cur_top = hex(int(cur_top,16) - i*1024)
        if(res[1][0]=="0x0"):
            break
        fees = fees + res[1]
        print("fees array length: ", len(fees))
        print("current block: ", int(cur_top,16))
    print("Done!", int(cur_top,16), int(LATEST_BLOCK,16), (int(LATEST_BLOCK,16) - int(cur_top,16)))
    return fees

```

STEP 0: SAVE THE DATA TO A CSV

```python
import csv

# Define the file path
file_path = './data/dl_fees.csv'

# Write the dl_fees_int array to the CSV file
with open(file_path, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(dl_fees_int)

```

ALGO 1: TWAP CALCULATION

```python
def downsample_array(arr, block_interval=1000, sample_rate=32):
    downsampled = []

    for start in range(0, len(arr), block_interval):
        end = start + block_interval
        chunk = arr[start:end]

        for i in range(0, len(chunk), sample_rate):
            sample = chunk[i:i + sample_rate]
            average = sum(sample) / len(sample) if sample else 0
            downsampled.append(average)

    return downsampled
```

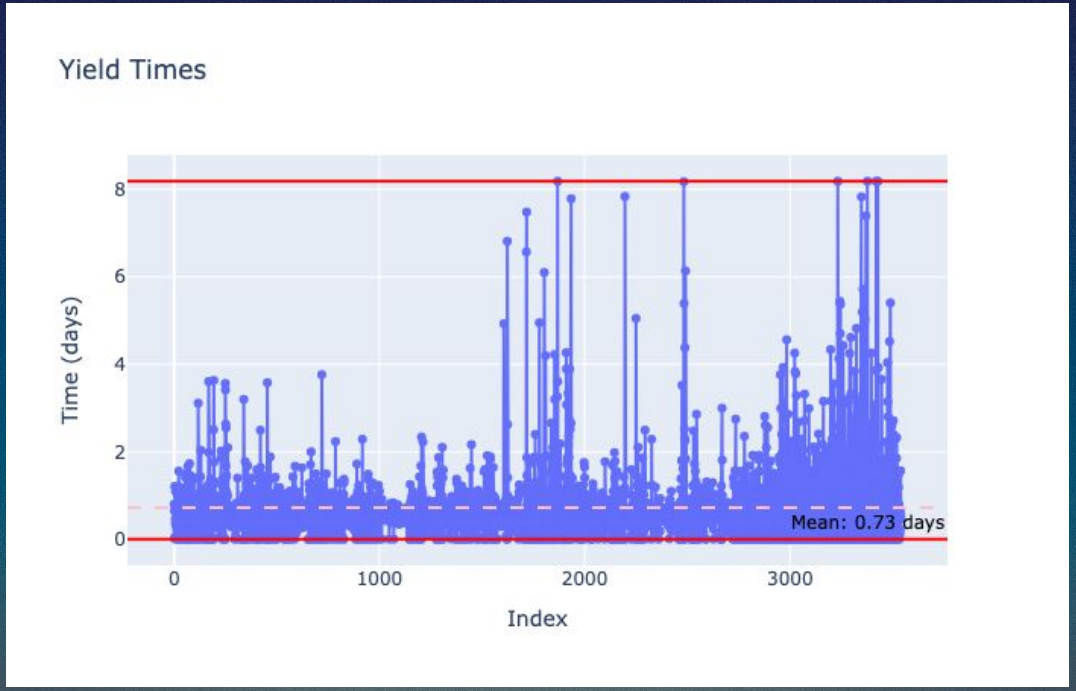
ALGO 2: FROM TWAP'S TO YIELD TIMES

```python
def map_delta_to_yield_time(delta, max_delta=MAX_DELTA, min_delta=MIN_DELTA, min_yield_time=MIN_YIELD_TIME, max_yield_time=MAX_YIELD_TIME):

    # ensure delta is within the specified range
    if delta < min_delta:
        delta = min_delta
    elif delta > max_delta:
        delta = max_delta

    # linear interpolation formula
    # y = y1 + (x - x1) * ((y2 - y1) / (x2 - x1))
    #yield_time = min_yield_time + (delta - min_delta) * ((max_yield_time - min_yield_time) / (max_delta - min_delta))

    #inverse
    yield_time = max_yield_time - (delta - min_delta) * ((max_yield_time - min_yield_time) / (max_delta - min_delta))

    return yield_time


def generate_yield_times(downsampled_data, max_delta=152e9, min_delta=-116e9, min_yield_time=MIN_YIELD_TIME, max_yield_time=MAX_YIELD_TIME):

    deltas = [downsampled_data[i] - downsampled_data[i - 1] for i in range(1, len(downsampled_data))]
    yield_times = [map_delta_to_yield_time(element, max_delta, min_delta, min_yield_time, max_yield_time) for element in deltas]
    return yield_times

def fees_to_yields(fees, sample_rate, block_interval, min_yield_time_, max_yield_time_, min_delta_, max_delta_):

    downsampled = downsample_array(fees, period=block_interval, sample_size=sample_rate)
    yield_times = generate_yield_times(downsampled, min_yield_time=min_yield_time_, max_yield_time=max_yield_time_, min_delta=min_delta_, max_delta=max_delta_)
    return yield_times

```

SO, WHAT NOW

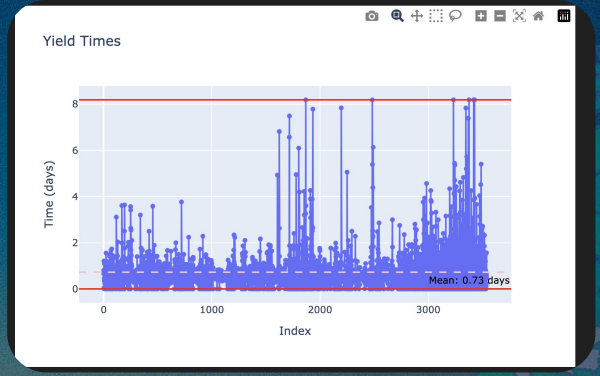
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123,
456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456,
789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678,
901234, 567890, 234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890,
234567, 890123, 456789, 123456, 789012, 345678, 901234, 567890, 234567, 890123

VISUALIZATION

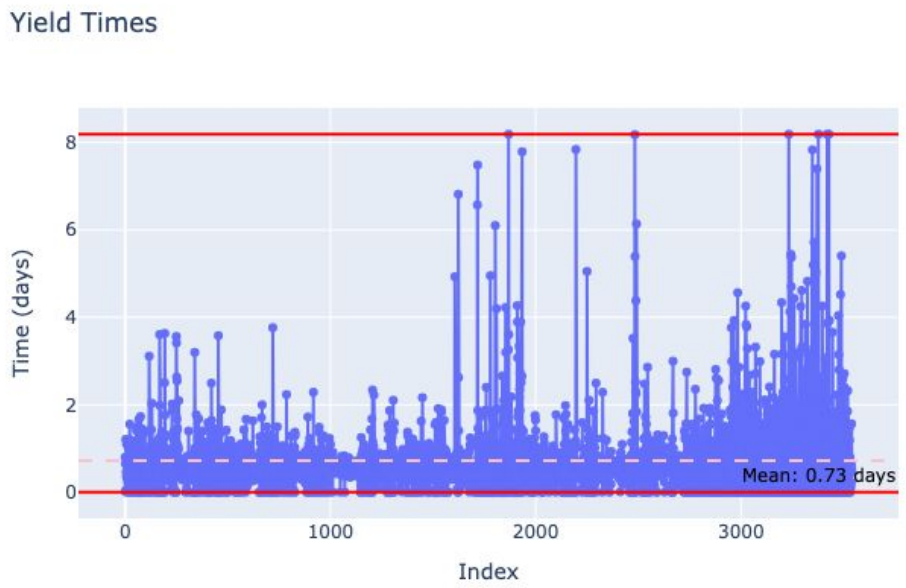


VISUALIZATION

plotly



VISUALIZATION

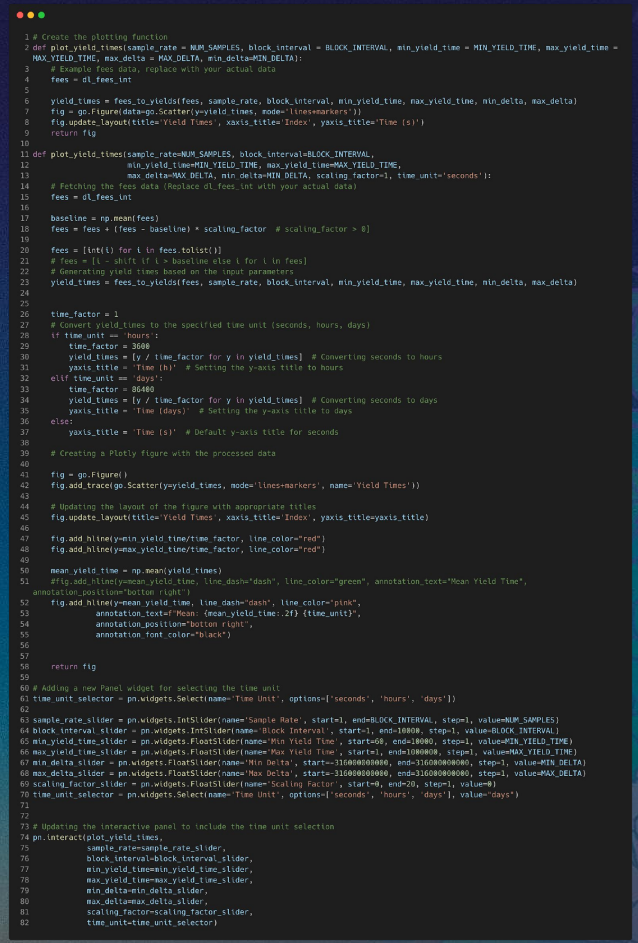



```python
# Create the plotting function
def plot_yield_times(sample_rate = NUM_SAMPLES, block_interval = BLOCK_INTERVAL, min_yield_time = MIN_YIELD_TIME, max_yield_time = MAX_YIELD_TIME, max_delta = MAX_DELTA, min_delta=MIN_DELTA):
    # Example fees data, replace with your actual data
    fees = dl_fees_int

    yield_times = fees_to_yields(fees, sample_rate, block_interval, min_yield_time, max_yield_time, min_delta, max_delta)
    fig = go.Figure(data=go.Scatter(y=yield_times, mode='lines+markers'))
    fig.update_layout(title='Yield Times', xaxis_title='Index', yaxis_title='Time (s)')
    return fig

def plot_yield_times(sample_rate=NUM_SAMPLES, block_interval=BLOCK_INTERVAL,
                     min_yield_time=MIN_YIELD_TIME, max_yield_time=MAX_YIELD_TIME,
                     max_delta=MAX_DELTA, min_delta=MIN_DELTA, scaling_factor=1, time_unit='seconds'):
    # Fetching the fees data (Replace dl_fees_int with your actual data)
    fees = dl_fees_int

    baseline = np.mean(fees)
    fees = fees + (fees - baseline) * scaling_factor  # scaling_factor > 0

    fees = [int(i) for i in fees.tolist()]
    # fees = [i - shift if i > baseline else i for i in fees]
    # Generating yield times based on the input parameters
    yield_times = fees_to_yields(fees, sample_rate, block_interval, min_yield_time, max_yield_time, min_delta, max_delta)


    time_factor = 1
    # Convert yield_times to the specified time unit (seconds, hours, days)
    if time_unit == 'hours':
        time_factor = 3600
        yield_times = [y / time_factor for y in yield_times]  # Converting seconds to hours
        yaxis_title = 'Time (h)'  # Setting the y-axis title to hours
    elif time_unit == 'days':
        time_factor = 86400
        yield_times = [y / time_factor for y in yield_times]  # Converting seconds to days
        yaxis_title = 'Time (days)'  # Setting the y-axis title to days
    else:
        yaxis_title = 'Time (s)'  # Default y-axis title for seconds

    # Creating a Plotly figure with the processed data

    fig = go.Figure()
    fig.add_trace(go.Scatter(y=yield_times, mode='lines+markers', name='Yield Times'))

    # Updating the layout of the figure with appropriate titles
    fig.update_layout(title='Yield Times', xaxis_title='Index', yaxis_title=yaxis_title)

    fig.add_hline(y=min_yield_time/time_factor, line_color="red")
    fig.add_hline(y=max_yield_time/time_factor, line_color="red")

    mean_yield_time = np.mean(yield_times)
    #fig.add_hline(y=mean_yield_time, line_dash="dash", line_color="green", annotation_text="Mean Yield Time",
  annotation_position="bottom right")
    fig.add_hline(y=mean_yield_time, line_dash="dash", line_color="pink",
                  annotation_text=f"Mean: {mean_yield_time:.2f} {time_unit}",
                  annotation_position="bottom right",
                  annotation_font_color="black")


    return fig

# Adding a new Panel widget for selecting the time unit
time_unit_selector = pn.widgets.Select(name='Time Unit', options=['seconds', 'hours', 'days'])

sample_rate_slider = pn.widgets.IntSlider(name='Sample Rate', start=1, end=BLOCK_INTERVAL, step=1, value=NUM_SAMPLES)
block_interval_slider = pn.widgets.IntSlider(name='Block Interval', start=1, end=18000, step=1, value=BLOCK_INTERVAL)
min_yield_time_slider = pn.widgets.FloatSlider(name='Min Yield Time', start=60, end=18000, step=1, value=MIN_YIELD_TIME)
max_yield_time_slider = pn.widgets.FloatSlider(name='Max Yield Time', start=1, end=1080000, step=1, value=MAX_YIELD_TIME)
min_delta_slider = pn.widgets.FloatSlider(name='Min Delta', start=-3160800000000, end=3160800000000, step=1, value=MIN_DELTA)
max_delta_slider = pn.widgets.FloatSlider(name='Max Delta', start=-3160800000000, end=3160800000000, step=1, value=MAX_DELTA)
scaling_factor_slider = pn.widgets.FloatSlider(name='Scaling Factor', start=0, end=20, step=1, value=0)
time_unit_selector = pn.widgets.Select(name='Time Unit', options=['seconds', 'hours', 'days'], value="days")


# Updating the interactive panel to include the time unit selection
pn.interact(plot_yield_times,
            sample_rate=sample_rate_slider,
            block_interval=block_interval_slider,
            min_yield_time=min_yield_time_slider,
            max_yield_time=max_yield_time_slider,
            min_delta=min_delta_slider,
            max_delta=max_delta_slider,
            scaling_factor=scaling_factor_slider,
            time_unit=time_unit_selector)
```

VISUALIZATION

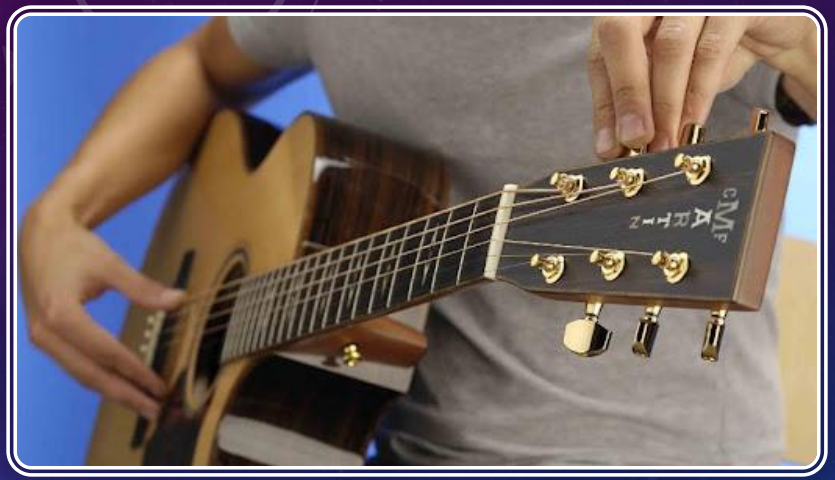
✳ Happy late night, Rico

Make Claude do it



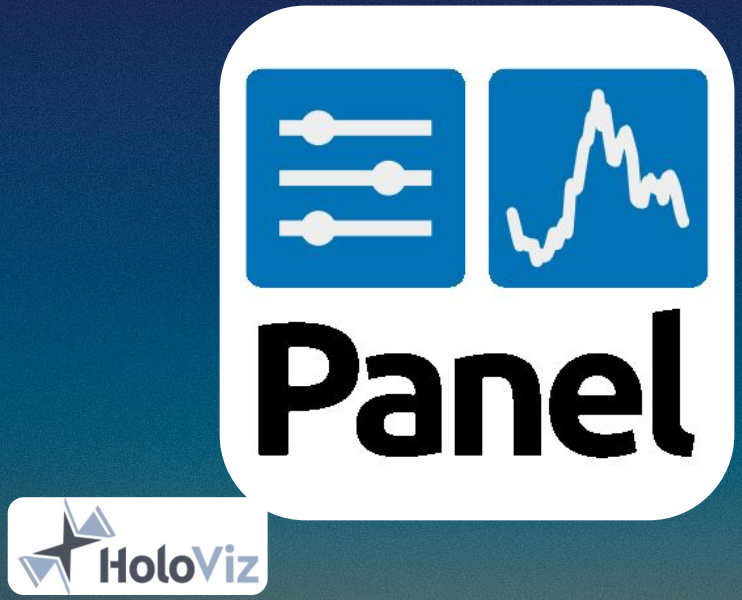
```python
# Create the plotting function
def plot_yield_times(sample_rate = NUM_SAMPLES, block_interval = BLOCK_INTERVAL, min_yield_time = MIN_YIELD_TIME, max_yield_time = MAX_YIELD_TIME, max_delta = MAX_DELTA, min_delta=MIN_DELTA):
    # Example fees data, replace with your actual data
    fees = dl_fees_int

    yield_times = fees_to_yields(fees, sample_rate, block_interval, min_yield_time, max_yield_time, min_delta, max_delta)
    fig = go.Figure(data=go.Scatter(y=yield_times, mode='lines+markers'))
    fig.update_layout(title='Yield Times', xaxis_title='Index', yaxis_title='Time (s)')
    return fig

def plot_yield_times(sample_rate=NUM_SAMPLES, block_interval=BLOCK_INTERVAL,
                     min_yield_time=MIN_YIELD_TIME, max_yield_time=MAX_YIELD_TIME,
                     max_delta=MAX_DELTA, min_delta=MIN_DELTA, scaling_factor=1, time_unit='seconds'):
    # Fetching the fees data (Replace dl_fees_int with your actual data)
    fees = dl_fees_int

    baseline = np.mean(fees)
    fees = fees + (fees - baseline) * scaling_factor  # scaling_factor > 0]

    fees = [int(i) for i in fees.tolist()]
    # fees = [i - shift if i > baseline else i for i in fees]
    # Generating yield times based on the input parameters
    yield_times = fees_to_yields(fees, sample_rate, block_interval, min_yield_time, max_yield_time, min_delta, max_delta)


    time_factor = 1
    # Convert yield_times to the specified time unit (seconds, hours, days)
    if time_unit == 'hours':
        time_factor = 3600
        yield_times = [y / time_factor for y in yield_times]  # Converting seconds to hours
        yaxis_title = 'Time (h)'  # Setting the y-axis title to hours
    elif time_unit == 'days':
        time_factor = 86400
        yield_times = [y / time_factor for y in yield_times]  # Converting seconds to days
        yaxis_title = 'Time (days)'  # Setting the y-axis title to days
    else:
        yaxis_title = 'Time (s)'  # Default y-axis title for seconds

    # Creating a Plotly figure with the processed data

    fig = go.Figure()
    fig.add_trace(go.Scatter(y=yield_times, mode='lines+markers', name='Yield Times'))

    # Updating the layout of the figure with appropriate titles
    fig.update_layout(title='Yield Times', xaxis_title='Index', yaxis_title=yaxis_title)

    fig.add_hline(y=min_yield_time/time_factor, line_color="red")
    fig.add_hline(y=max_yield_time/time_factor, line_color="red")

    mean_yield_time = np.mean(yield_times)
    #fig.add_hline(y=mean_yield_time, line_dash="dash", line_color="green", annotation_text="Mean Yield Time", annotation_position="bottom right")
    fig.add_hline(y=mean_yield_time, line_dash="dash", line_color="pink",
                  annotation_text=f"Mean: {mean_yield_time:.2f} {time_unit}",
                  annotation_position="bottom right",
                  annotation_font_color="black")


    return fig

# Adding a new Panel widget for selecting the time unit
time_unit_selector = pn.widgets.Select(name='Time Unit', options=['seconds', 'hours', 'days'])

sample_rate_slider = pn.widgets.IntSlider(name='Sample Rate', start=1, end=BLOCK_INTERVAL, step=1, value=NUM_SAMPLES)
block_interval_slider = pn.widgets.IntSlider(name='Block Interval', start=1, end=18000, step=1, value=BLOCK_INTERVAL)
min_yield_time_slider = pn.widgets.FloatSlider(name='Min Yield Time', start=60, end=18000, step=1, value=MIN_YIELD_TIME)
max_yield_time_slider = pn.widgets.FloatSlider(name='Max Yield Time', start=1, end=1080000, step=1, value=MAX_YIELD_TIME)
min_delta_slider = pn.widgets.FloatSlider(name='Min Delta', start=-3160000000000, end=3160000000000, step=1, value=MIN_DELTA)
max_delta_slider = pn.widgets.FloatSlider(name='Max Delta', start=-3160000000000, end=3160000000000, step=1, value=MAX_DELTA)
scaling_factor_slider = pn.widgets.FloatSlider(name='Scaling Factor', start=0, end=20, step=1, value=0)
time_unit_selector = pn.widgets.Select(name='Time Unit', options=['seconds', 'hours', 'days'], value="days")


# Updating the interactive panel to include the time unit selection
pn.interact(plot_yield_times,
            sample_rate=sample_rate_slider,
            block_interval=block_interval_slider,
            min_yield_time=min_yield_time_slider,
            max_yield_time=max_yield_time_slider,
            min_delta=min_delta_slider,
            max_delta=max_delta_slider,
            scaling_factor=scaling_factor_slider,
            time_unit=time_unit_selector)
```

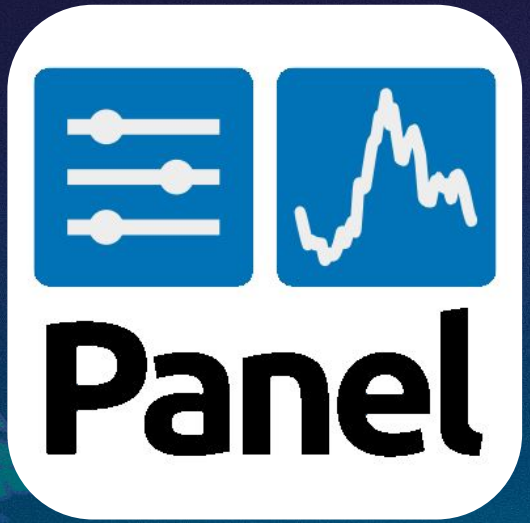
```
NUM_SAMPLES = 32
BLOCK_INTERVAL = 1000
MIN_YIELD_TIME = 387
MAX_YIELD_TIME = 707412
MIN_DELTA = -54067095249
MAX_DELTA = 3870911308
OBSERVATION_PERIOD = BLOCK_INTERVAL*2*365
```

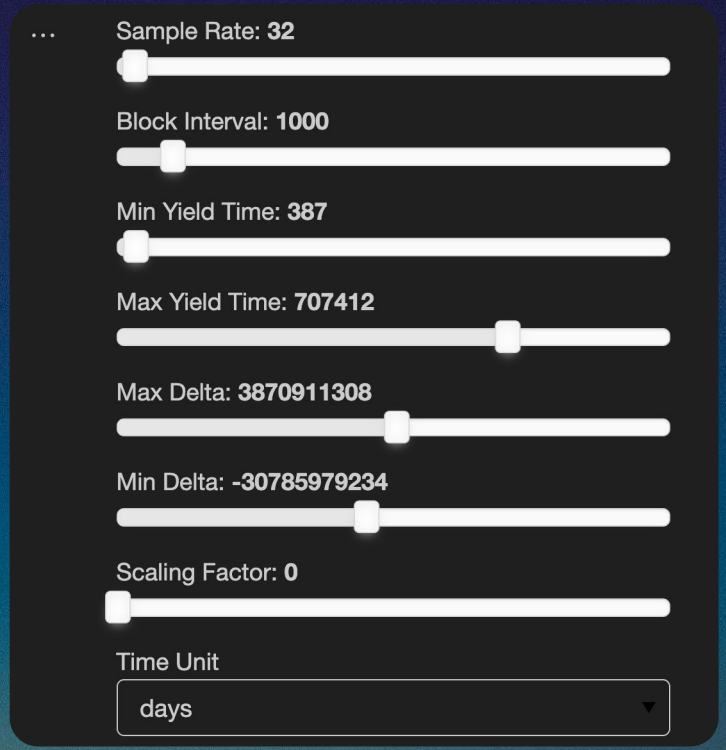


TUNING YOUR SYSTEM (PICKING THE PARAMETERS / VARIABLES)

ENTER... PANEL




Panel


Sample Rate: 32
Block Interval: 1000
Min Yield Time: 387
Max Yield Time: 707412
Max Delta: 3870911308
Min Delta: -30785979234
Scaling Factor: 0
Time Unit
days


jupyter

Sample Rate: **32**

Block Interval: **1000**

Min Yield Time: **387**

Max Yield Time: **707412**

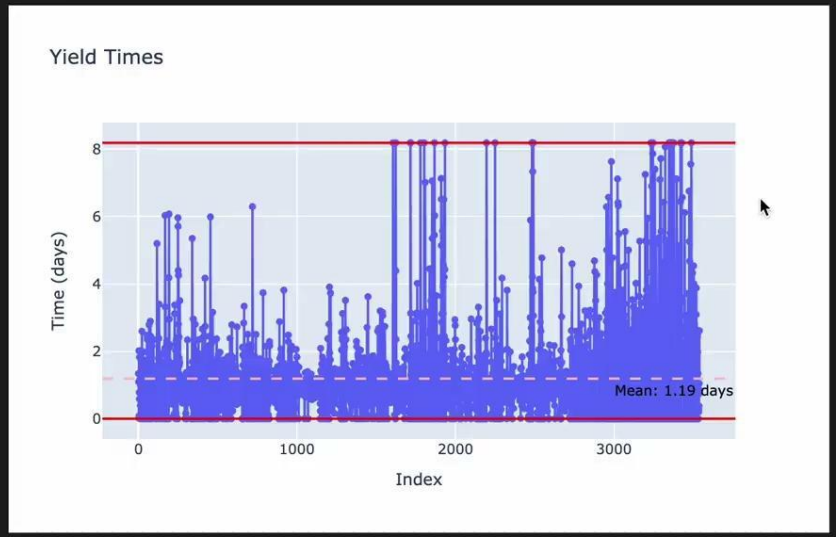
Max Delta: **3870911308**

Min Delta: **-30785979234**

Scaling Factor: **0**

Time Unit

days

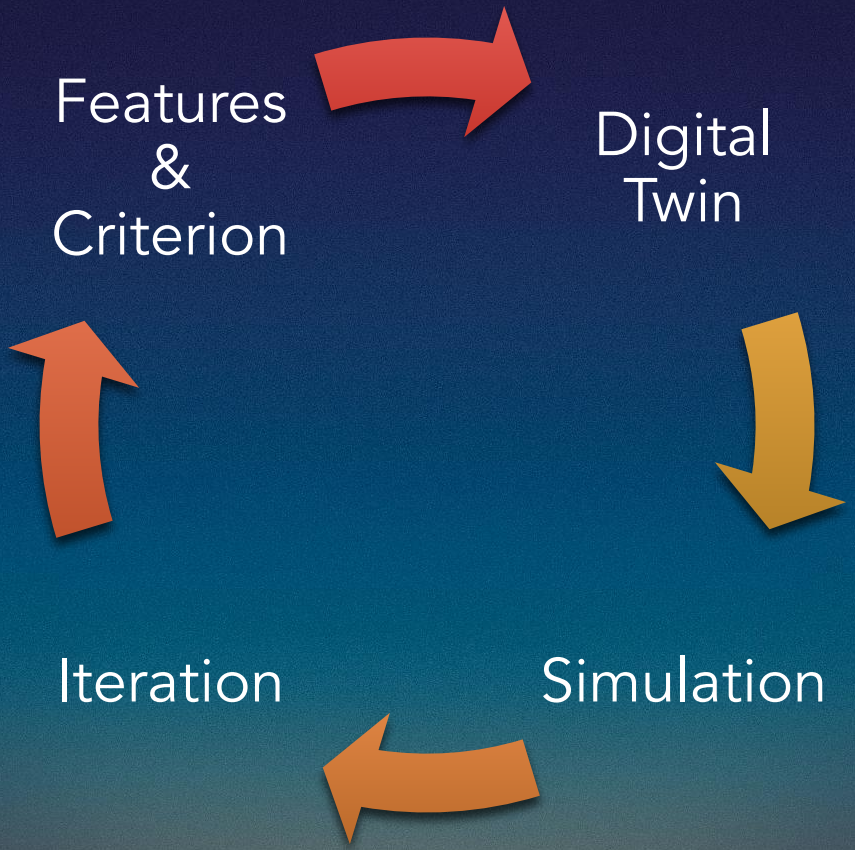


+ INTEGRATION AND A | B TESTING

Implementation

A | B Testing with Digital Twin

Take this away with you



Congratulations!
You have now officially made a complex system!

You are officially smart

INSPIRED BY

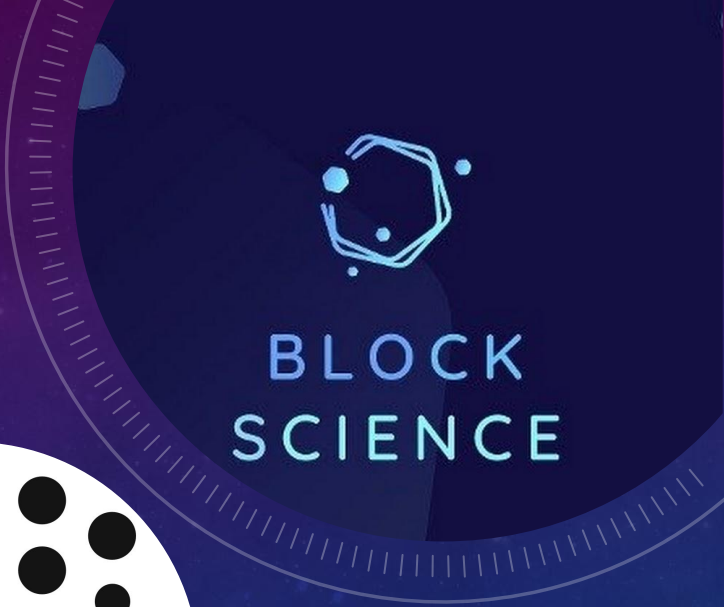
Trent McConaghy's:

[Ocean Market Balancer Simulations For TE Academy]

Shawn Anderson and Longtail Financial

The Token Engineering Academy

BlockScience



FURTHER READING

THANK YOU

Nico Rodriguez
Creator of CafeCosmos
x.com/CafeCosmosHQ