

Chasing nanoseconds: data science for low latency networks

Omer Yuksel, Performance Analyst

IMC Trading

Introduction - IMC

- Technology-driven trading
- Amsterdam, Chicago, Sydney



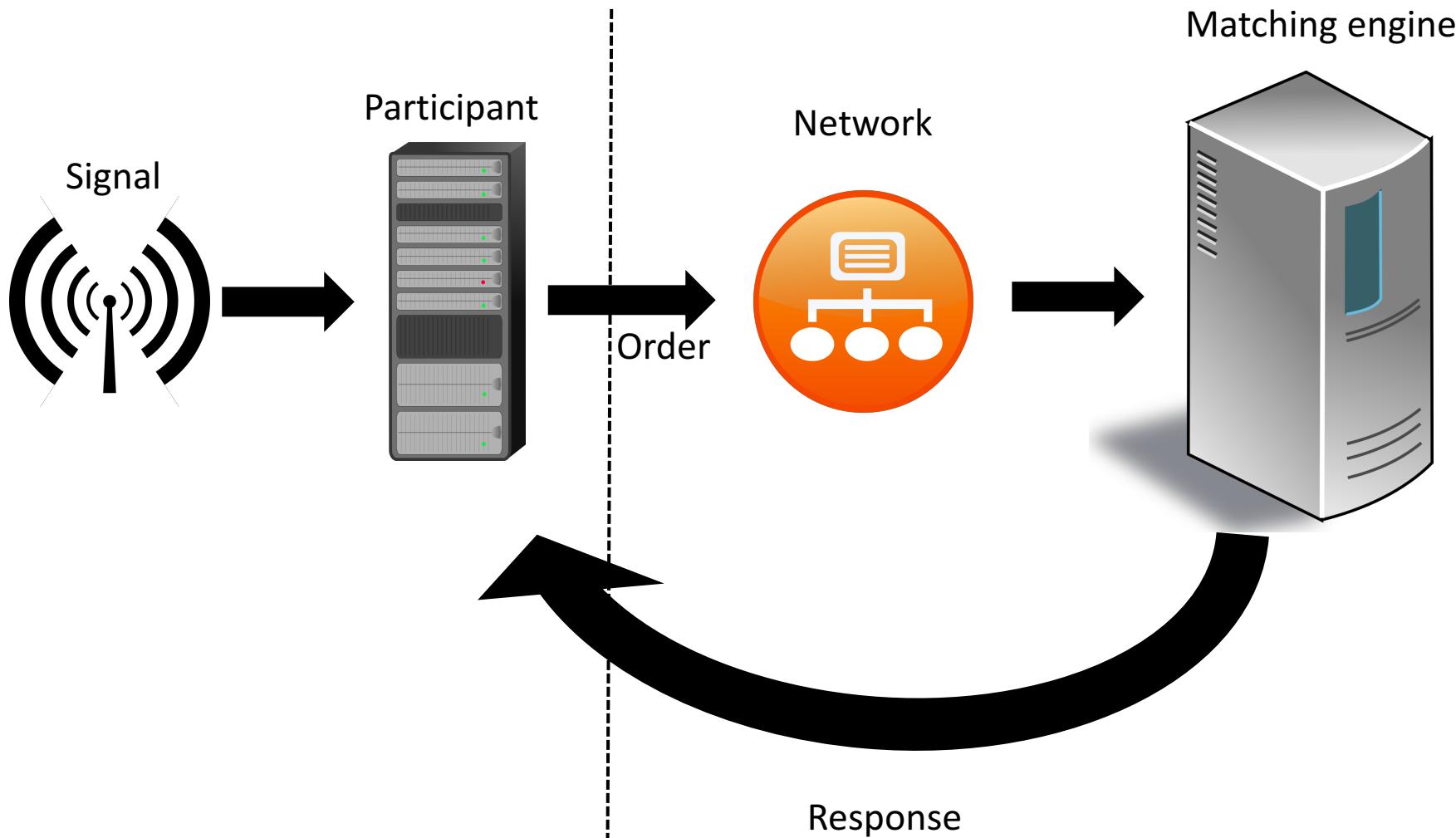
www.imc.com

Outline

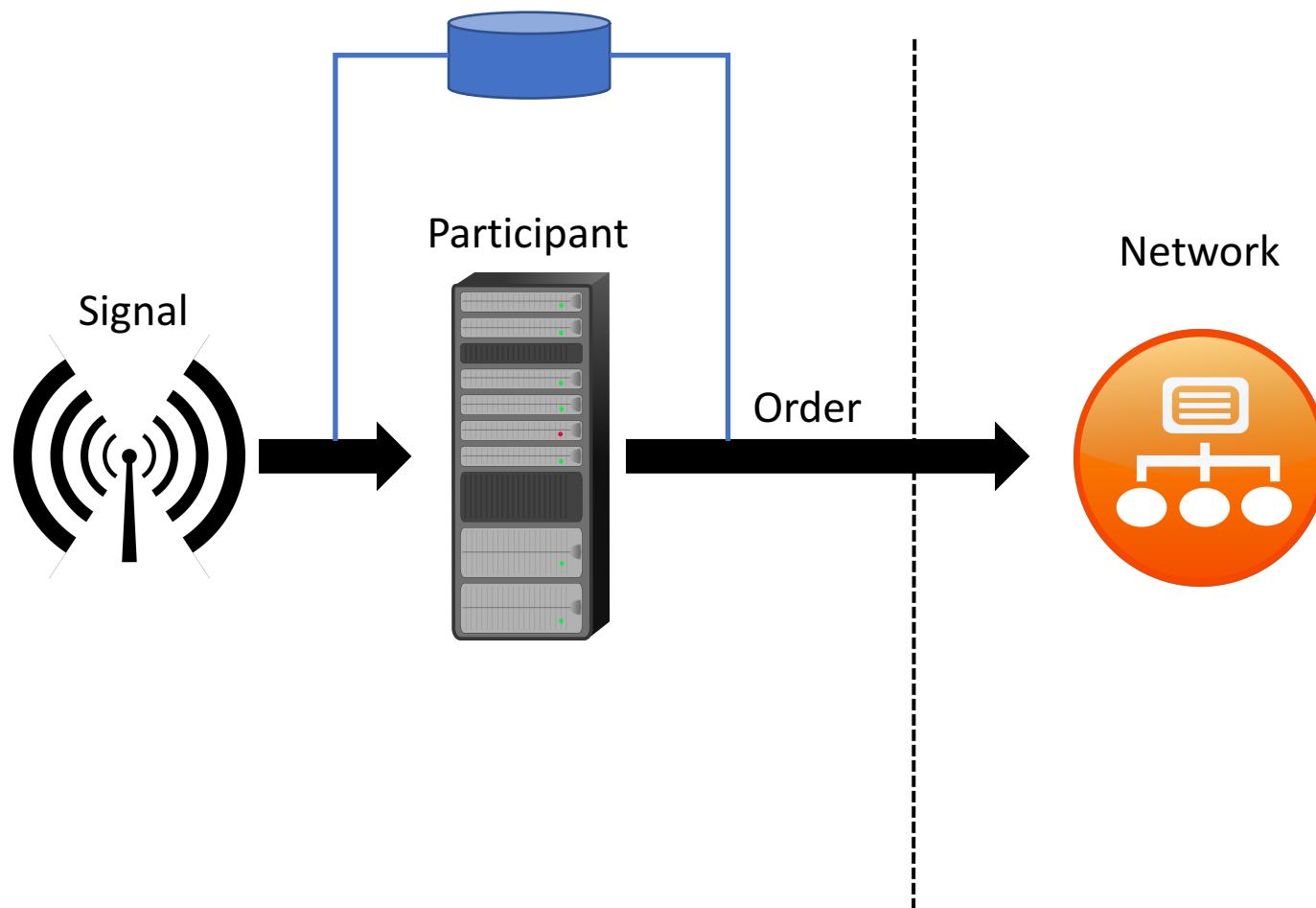
- Trading loop
- Modelling
- Anomaly Detection
- Challenges and pitfalls

Trading

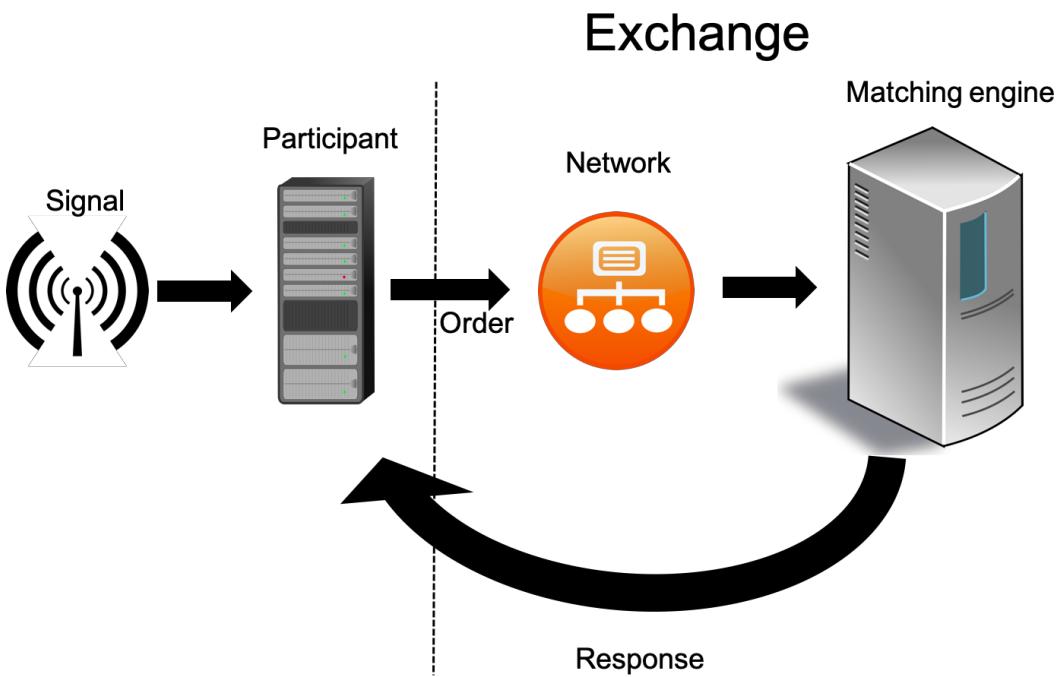
Exchange



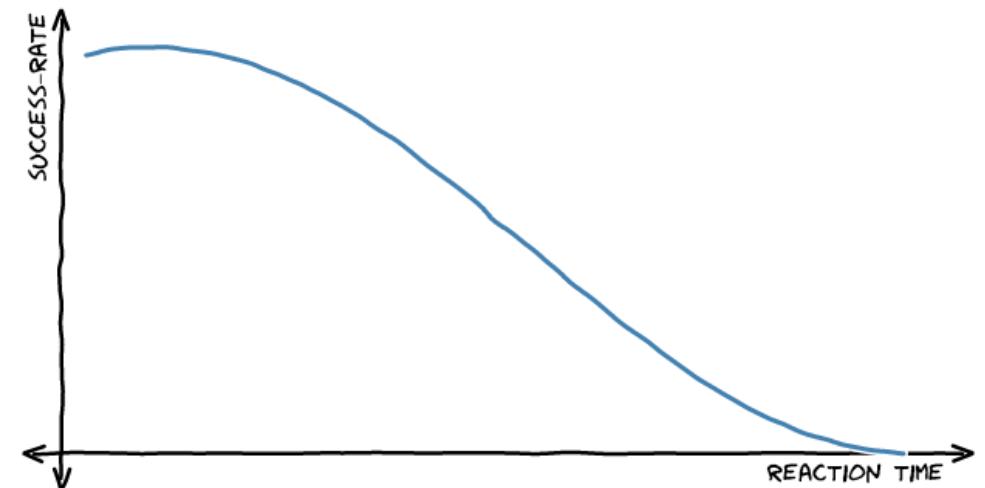
Reaction Time



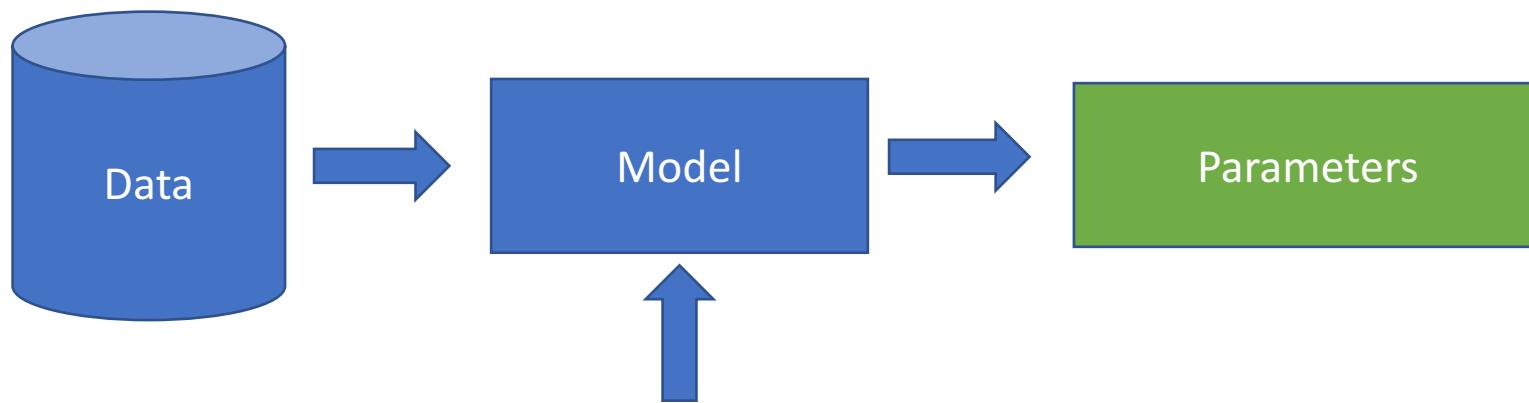
Modelling



- How often are we queuing?
- How much does queuing matter?
- How much should we improve our reaction time?



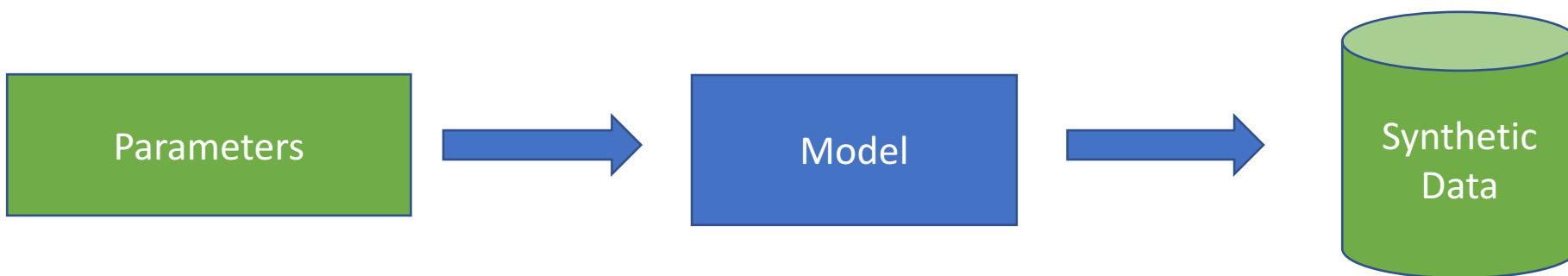
Modelling overview



	ts	reaction_time	time_to_me	success	total_time_to_me
0	2019-06-03 09:00:00.042956193+02:00	606	5085	False	5691
1	2019-06-03 09:00:00.207588123+02:00	647	4500	False	5147
2	2019-06-03 09:00:00.276477343+02:00	633	4469	True	5102
3	2019-06-03 09:00:00.306361141+02:00	577	4459	True	5036
4	2019-06-03 09:00:00.389016054+02:00	599	4349	True	4948
5	2019-06-03 09:00:00.426910915+02:00	578	4515	True	5093
6	2019-06-03 09:00:00.552713100+02:00	572	4472	True	5044
7	2019-06-03 09:00:00.783978162+02:00	566	4680	True	5246
8	2019-06-03 09:00:00.870361884+02:00	606	4457	True	5063
9	2019-06-03 09:00:00.948479263+02:00	563	4394	True	4957

- Minimum queuing delay: 250ns
- Number of switches: 2

Modelling overview



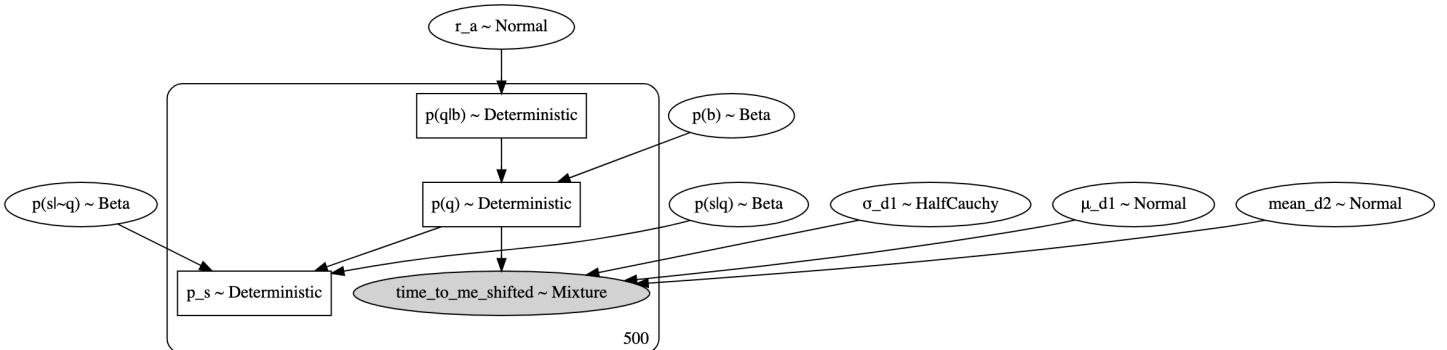
- Model validation
- What-if scenarios

Approaches

- Simulate all details
 - SimPy

```
class FIFOswitch(NetworkNode):  
    def __init__(self, name, env, destination, time_to_destination):=■  
  
    def run(self):  
        while True:  
            packet = yield self.incoming_packets.get()  
            self.debug(f'{packet} arrived')  
            self.env.process(self.process_packet(packet))  
  
    def process_packet(self, packet):  
        yield self.env.timeout(SWITCH_PROCESSING_DELAY)  
        self.outgoing_packets.put(packet)  
  
    def transmit_outgoing_packets(self):  
        while True:  
            packet = yield self.outgoing_packets.get()  
            self.debug(■)  
            self.send_to_destination(packet)  
            yield (self.env.timeout(TRANSMISSION_DELAY))
```

- Formulate as probability distributions
 - PyMC3



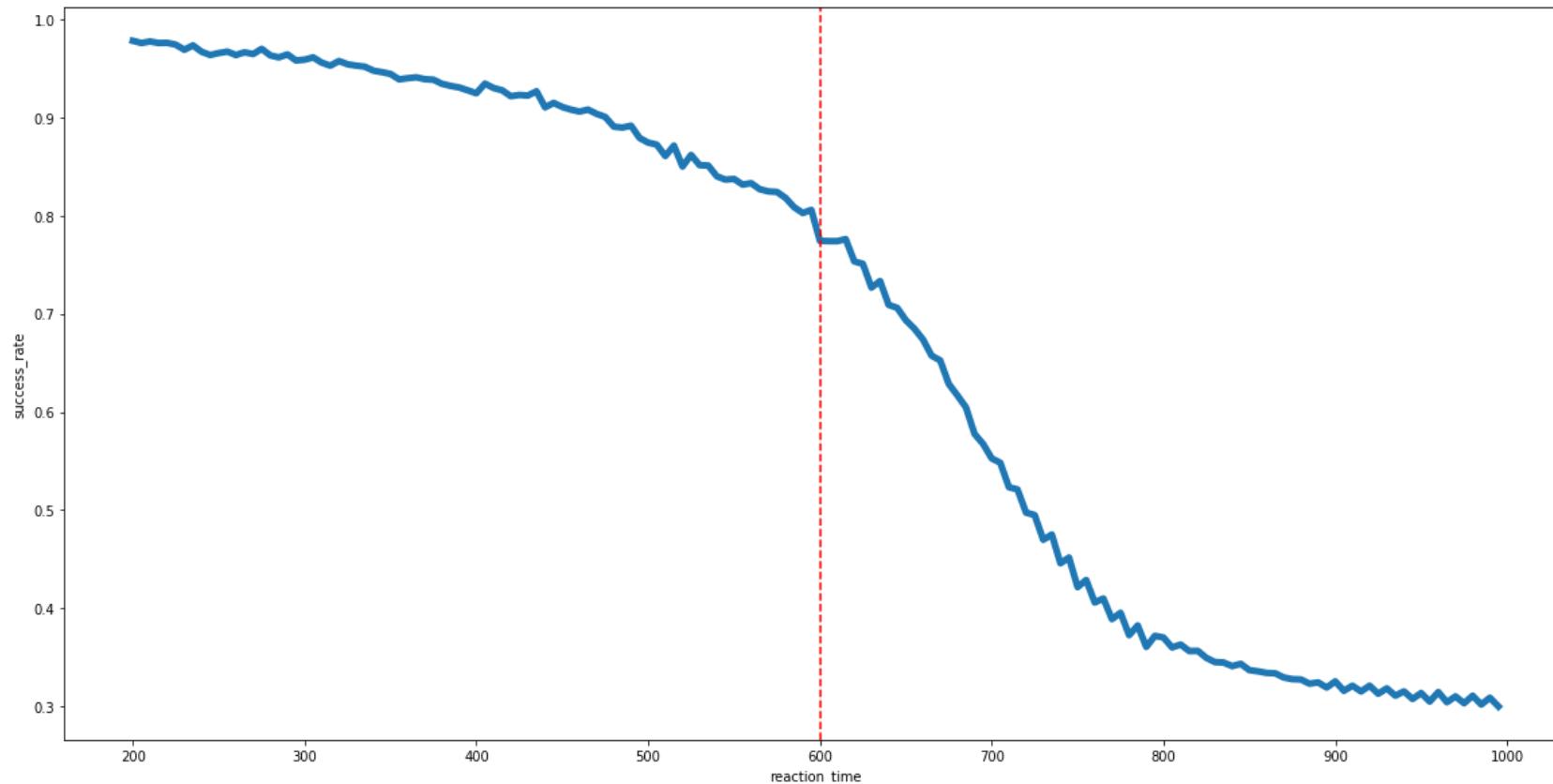
Model output (I)

Model parameters

		mean	sd	mc_error	hpd_2.5	hpd_97.5
Reaction time	r_mu	599.846909	1.566378	0.016985	596.739331	602.869329
	r_sigma	22.180482	0.772145	0.008591	20.637166	23.673577
Queuing	p_q	0.191285	0.029576	0.000252	0.135996	0.251843
Bursts	p_b	0.255693	0.028668	0.000265	0.201732	0.313731
Success rate	p_s	0.788471	0.024746	0.000210	0.738779	0.835984
Base success rate	p_s_a	0.424210	0.039147	0.000366	0.349996	0.502416

Model output (II)

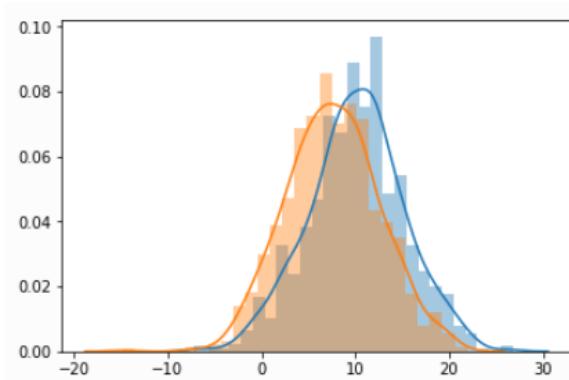
Average reaction time vs success



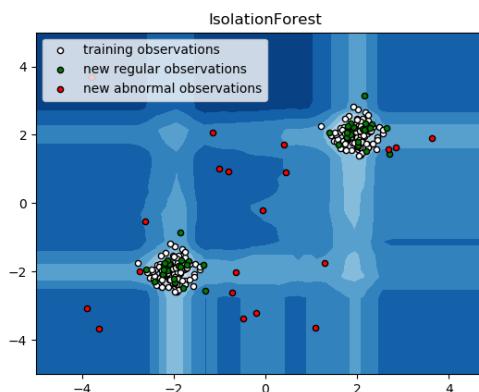
Anomaly Detection

Detecting unusual events without training on prior examples.

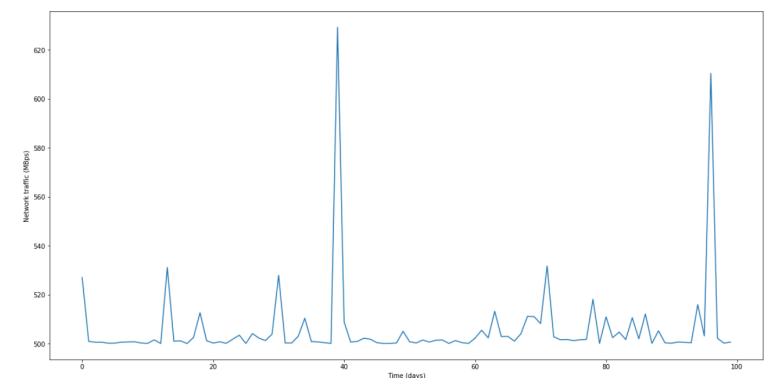
Probabilistic



Machine Learning



Time-series based



Anomaly detection at IMC

Centralized

- Domain-agnostic
- 1-dimensional time-series
- Java-based, with Python extensions
- Focus on interpretability and alerting

Customized

- Domain-specific
- Python-based
- Focus on flexibility

Example metrics:

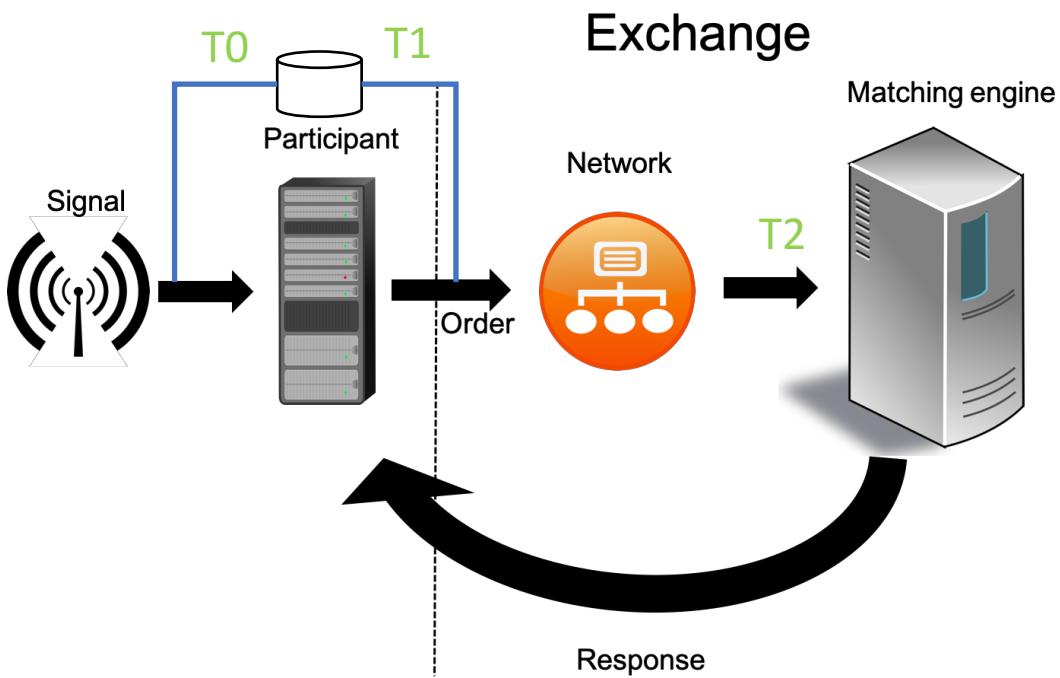
- Reaction time
- Model parameters
- Clock synchronization

Lessons learned

- Reporting >> algorithm performance
- Think about the type of anomalies to detect
- Do as much as possible at preprocessing step
- Scale alerts to the size of the organization

What could go wrong?

Sources of noise



- Propagation delay
- Clock synchronization

Using nanosecond timestamps – Pandas (I)

Attempt #1

```
nano_df = pd.read_csv(FILE)
nano_df = nano_df.dropna().astype(int)

nano_df['reaction_time'] = nano_df['t1'] - nano_df['t0']
```

```
nano_df.sample(5)
```

	t0	t1	t2	t3	product_id	quantity	side	reaction_time
98333	1570187623063325440	1570187623063326720	1570187623063338075	1570187623064082201	18	6	1	1280
222264	1570181872005868032	1570181872005869312	1570181872005857293	1570181872005907126	31	5	2	1280
196401	1570192204183381248	1570192204183382528	1570192204183382834	1570192204183438711	3	1	2	1280
59639	1570199323832303104	1570199323832320256	1570199323832267033	1570199323832390999	3	1	2	17152
169307	1570178421201831424	1570178421201832704	1570178421201771461	1570178421201886304	29	3	1	1280

Using nanosecond timestamps – Pandas (II)

```
nano_df.reaction_time.value_counts()
```

```
1280      155871
1536       37902
17152      26768
16896      19845
17408       1487
16640        493
1024        457
17664         2
Name: reaction_time, dtype: int64
```

Using nanosecond timestamps – Pandas (III)

```
In [36]: (nano_df.reaction_time % 256).value_counts()
```

```
Out[36]: 0    242825  
          Name: reaction_time, dtype: int64
```

```
In [37]: (nano_df.t0 % 256).value_counts()
```

```
Out[37]: 0    242825  
          Name: t0, dtype: int64
```

Float64 storage

- Trade-off precision vs range
- Float64 data type: sign bit, 53 bits mantissa, 11 bits exponent
- Bits required to represent nanosecond timestamps now: 61

Load as object

```
nano_df = pd.read_csv(FILE, dtype='object')
nano_df = nano_df.dropna().astype(int)

nano_df['reaction_time'] = nano_df['t1'] - nano_df['t0']
```

```
nano_df.reaction_time.value_counts().sort_values()
```

```
17352      1
1437       1
17393      1
17392      1
17431      1
...
1334      2588
1331      2601
1326      2603
1332      2606
1327      2639
Name: reaction_time, Length: 918, dtype: int64
```

Load as nullable int (|)

```
nano_df = pd.read_csv(FILE, dtype={'t0': 'Int64', 't1': 'Int64'})  
  
nano_df['reaction_time'] = nano_df['t1'] - nano_df['t0']  
  
nano_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 344949 entries, 0 to 344948  
Data columns (total 8 columns):  
t0            242825 non-null Int64  
t1            242825 non-null Int64  
t2            344949 non-null int64  
t3            344949 non-null int64  
product_id    344949 non-null int64  
quantity      344949 non-null int64  
side          344949 non-null int64  
reaction_time 242825 non-null Int64  
dtypes: Int64(3), int64(5)  
memory usage: 22.0 MB
```

Load as nullable int (II)

```
nano_df.sample(5)
```

	t0	t1	t2	t3	product_id	quantity	side	reaction_time
184304	1570191755423981568	1570191755423998720	1570191755424178135	1570191755424253187	18	4	2	17152
171517	1570194430324996864	1570194430324998144	1570194430325009103	1570194430325807014	31	8	1	1280
39583	1570205505659457280	1570205505659458560	1570205505659300139	1570205505659462984	57	1	1	1280
229145	1570195867805028864	1570195867805030144	1570195867805055969	1570195867805143056	18	1	2	1280
126857	NaN	NaN	1570188729057005056	1570188729057072034	31	3	1	NaN

```
(nano_df['reaction_time'] % 256).value_counts()
```

```
0      242825  
Name: reaction_time, dtype: int64
```

Use converters – np.int64

```
def convert_to_int64(x):
    try:
        return np.int64(x)
    except ValueError:
        return 0

nano_df = pd.read_csv(FILE, converters={'t0': convert_to_int64, 't1': convert_to_int64})
nano_df = nano_df.query('t1>0 and t0>0')
nano_df['reaction_time'] = nano_df['t1'] - nano_df['t0']
```

```
nano_df.reaction_time.value_counts().sort_values()
```

```
17352      1
1437       1
17393      1
17392      1
17431      1
...
1334      2588
1331      2601
1326      2603
1332      2606
1327      2639
Name: reaction_time, Length: 918, dtype: int64
```

Using datetime type with lower precision

- Javascript: 54-bit timestamps
- Millisecond precision

How to retain nanosecond precision?

Retaining nanosecond precision

- Workaround #1: Cast to string for visualization purposes
- Workaround #2: try to map time of the day to 54-bit timestamps

Retaining nanosecond precision

```
t_int = 1570197332597561328
```

```
pd.to_datetime(t_int)
```

```
Timestamp('2019-10-04 13:55:32.597561328')
```

```
t_int_day = (t_int % (24*3600*10**9)) # Drop date
```

```
t_int, t_int_day
```

```
(1570197332597561328, 50132597561328)
```

```
np.log2(t_int), np.log(t_int_day)
```

```
(60.44565158738884, 31.545692562365183)
```

Retaining nanosecond precision

Converting t_int to datetime64 and t_int_day to datetime in milliseconds:

t_int: 2019-10-04 13:55:32.597561328

t_int_day: 3558-08-22 11:32:41.328

Conversion table

Micro	Actual
Milliseconds	1 ns
Seconds	1 us
Minutes	60 us
Hours	3.6 ms
Days	86.4 ms
Months	~2.5 seconds
Years	~30 seconds
Centuries	~8 hours

Questions?

Data pipelines

Capture/Timestamping device



Capture host



Kafka



Storage / Applications



Raw data

0000	00	16	3e	ba	b7	75	fe	7f	3a	e7	ad	1f	08	00	45	c0	...>..u..	:.....E.
0010	00	97	9e	75	40	00	40	06	81	20	0a	00	03	01	0a	00	...u@. @.
0020	03	0b	d9	64	19	e9	38	81	8a	62	4e	f9	e8	c1	80	18	...d..8.	.bN.....
0030	00	3c	1a	95	00	00	01	01	08	0a	00	01	75	90	00	01	.<.....u....
0040	75	00	01	00	00	00	00	00	00	00	00	00	00	00	00	00

PCAP

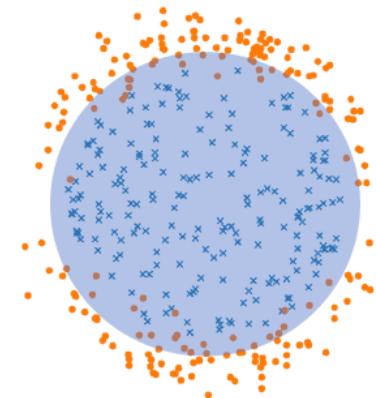
```
typedef struct pcaprec_hdr_s {  
    quint32 ts_sec;           /* timestamp seconds */  
    quint32 ts_usec;          /* timestamp microseconds */  
    quint32 incl_len;         /* number of octets of packet saved in file */  
    quint32 orig_len;          /* actual length of packet */  
} pcaprec_hdr_t;
```

Decoded Message



Classification vs Anomaly Detection

Classification



Anomaly detection

