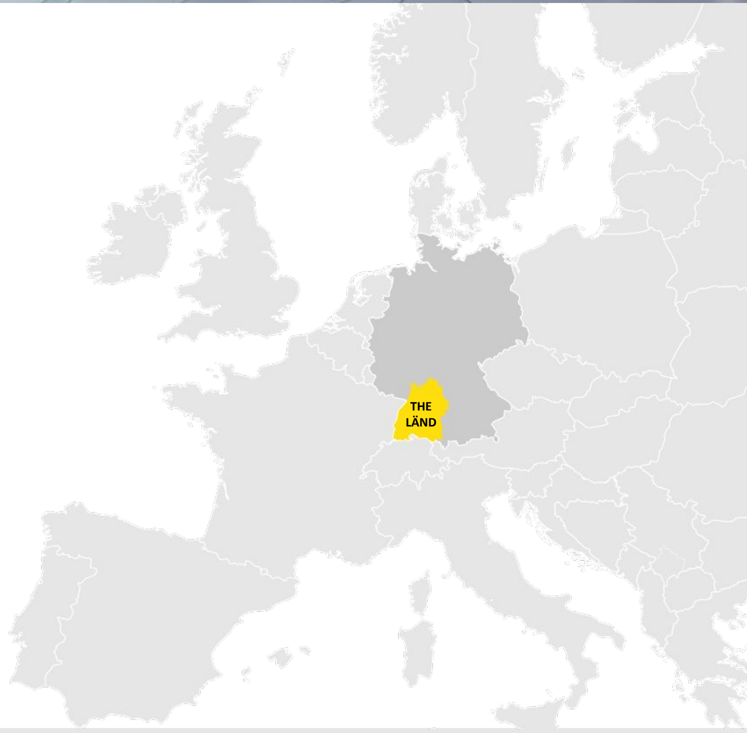
Flow Processing at BelWü

DENOG14

15th November 2022

Daniel Nägele

naegele@belwue.de



Purposes of Flow Processing

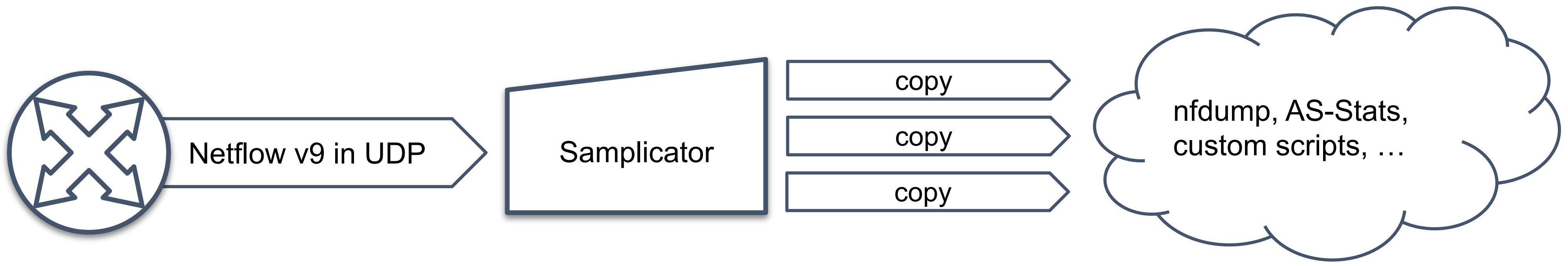
There are many, equally valid reasons to collect and process flows.

Some examples are:

- Find new, impactful peerings, e.g. regarding the utilization of some interface
- Derive flowspec rules for a customer receiving a DoS traffic
- Show awesome graphs to customers
- Detect devices that talk to known bad actors
- Answer this nagging one-off question about your traffic that someone came up with

Problems of Flow Processing

- Some applications require specific “vantage points” in a network
- Different formats and various hardware limitations
- What reality often looks like:
  - A Samplicator[1] instance, re-sending spoofed UDP datagrams containing flows
  - Different specialized tools parse the same flows



[1] https://github.com/sleinen/samplicator

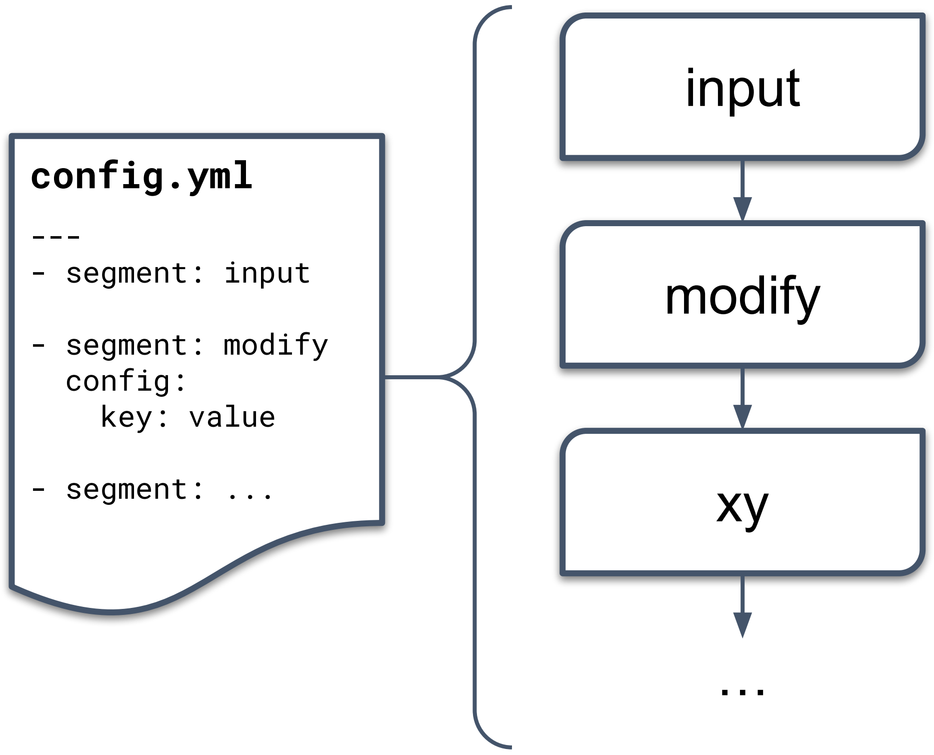
How does yet another tool solve anything?

Focus on “Application Layer” flow processing:

- Stream separation based on arbitrary criteria
- Enrich interesting flows only, drop noise early
- Full multi-tenant support for flow monitoring
- Unify flows from different sources with arbitrary granularity
- Fully reproducible yet extensible setups

flowpipeline Tooling

- Completely configuration-defined
- Single dependency-free binary
- Support for any common flow format
- Segments act on single, protobuf-encoded flow messages and pass them along
- Many different segments are available
- Open Source



config.yml

```
---
- segment: input

- segment: modify
  config:
    key: value

- segment: ...
```

Where do Flow Messages come from?

GoFlow 2

- use Goflow v2 to listen for flows in raw format
- supports network devices with sFlow, IPFIX or Netflow v9

Where do Flow Messages come from?

GoFlow2

- use Goflow v2 to listen for flows in raw format
- supports network devices with sFlow, IPFIX or Netflow v9

APACHE kafka

- receive flows generated by another flowpipeline from a Kafka cluster
- flows can be pre-filtered or pre-enriched
- ability to output to Kafka

Where do Flow Messages come from?

GoFlow 2

- use Goflow v2 to listen for flows in raw format
- supports network devices with sFlow, IPFIX or Netflow v9

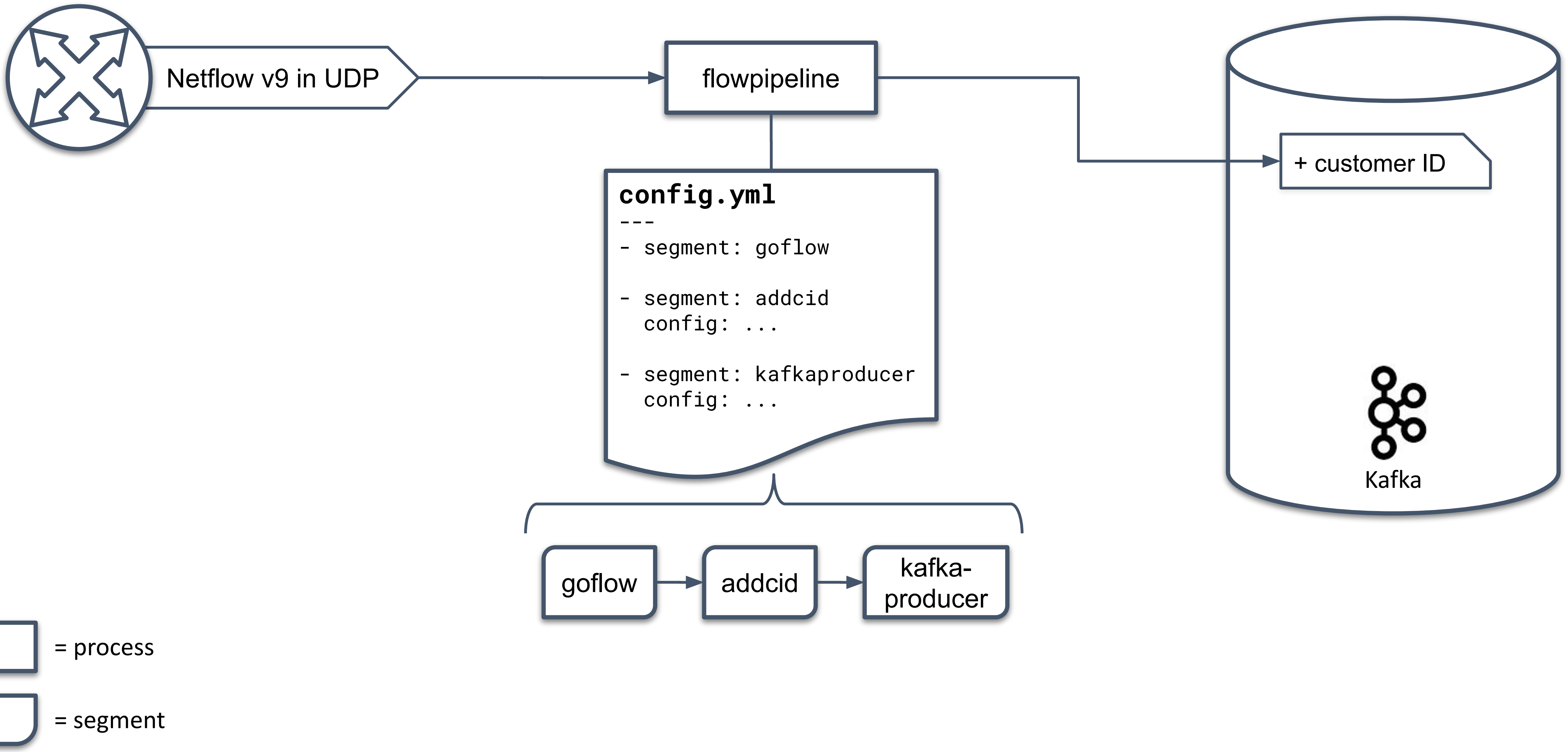
Apache Kafka

- receive flows generated by another flowpipeline from a Kafka cluster
- flows can be pre-filtered or pre-enriched
- ability to output to Kafka

eBPF

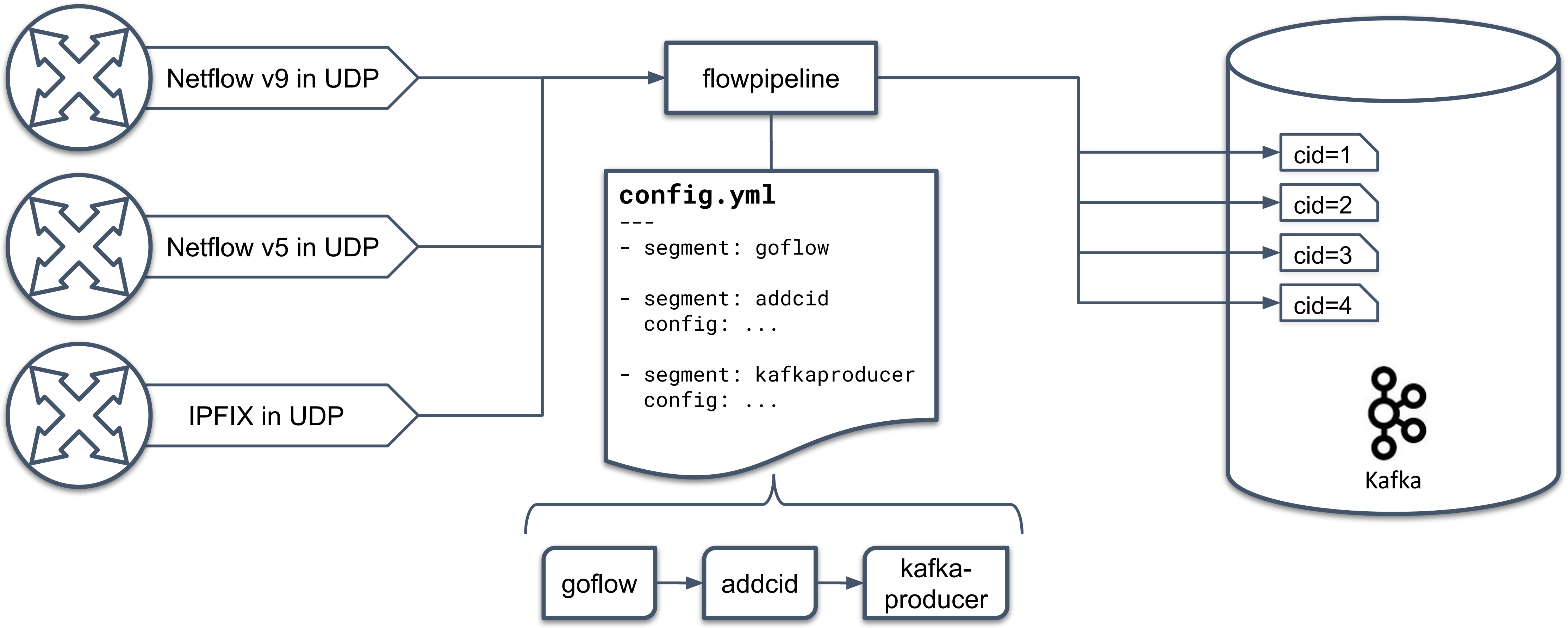
- use eBPF to dump packet headers
- match packets to flows in custom cache using 5-tuple
- additional information available (packet IAT, …)
- working, but still WIP

Routing Flow Messages



```
config.yml
---
- segment: goflow

- segment: addcid
  config: ...

- segment: kafkaproducer
  config: ...
```

Routing Flow Messages

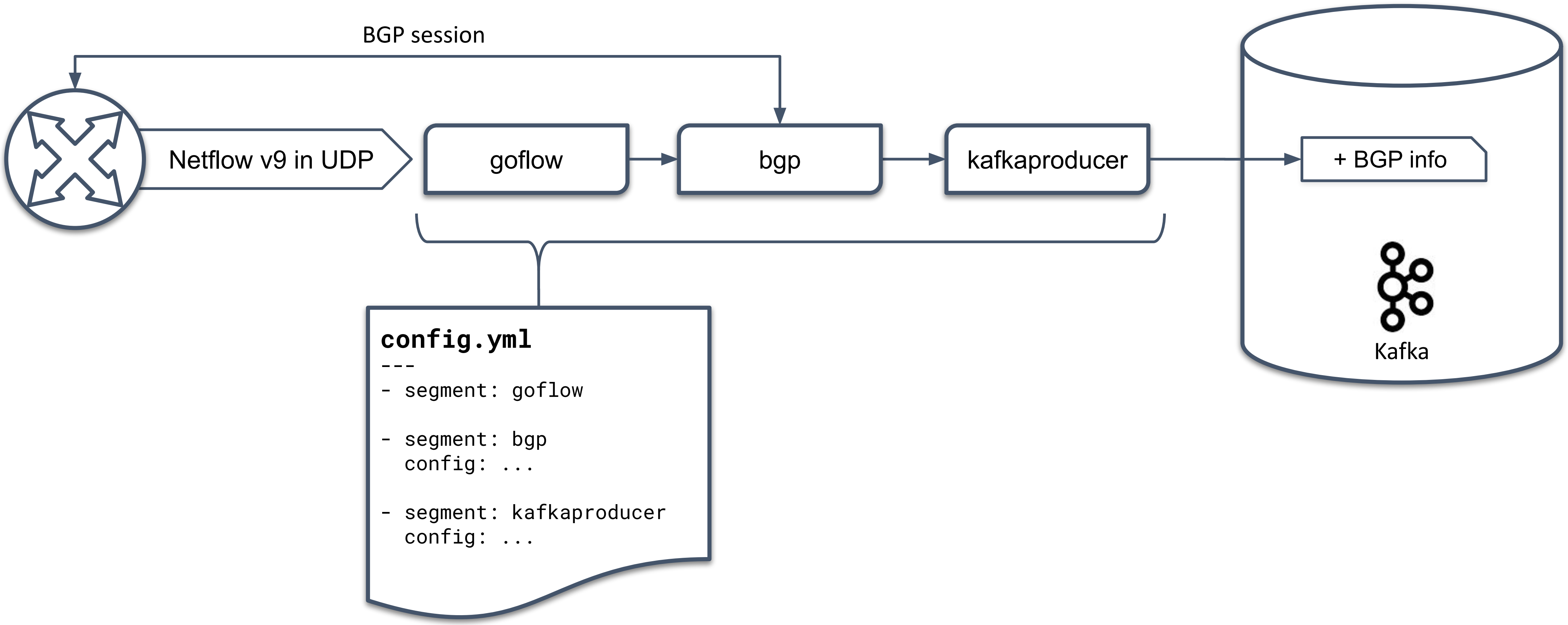


```
config.yml
---
- segment: goflow

- segment: addcid
  config: ...

- segment: kafkaproducer
  config: ...
```

Flow Enrichment and Modification

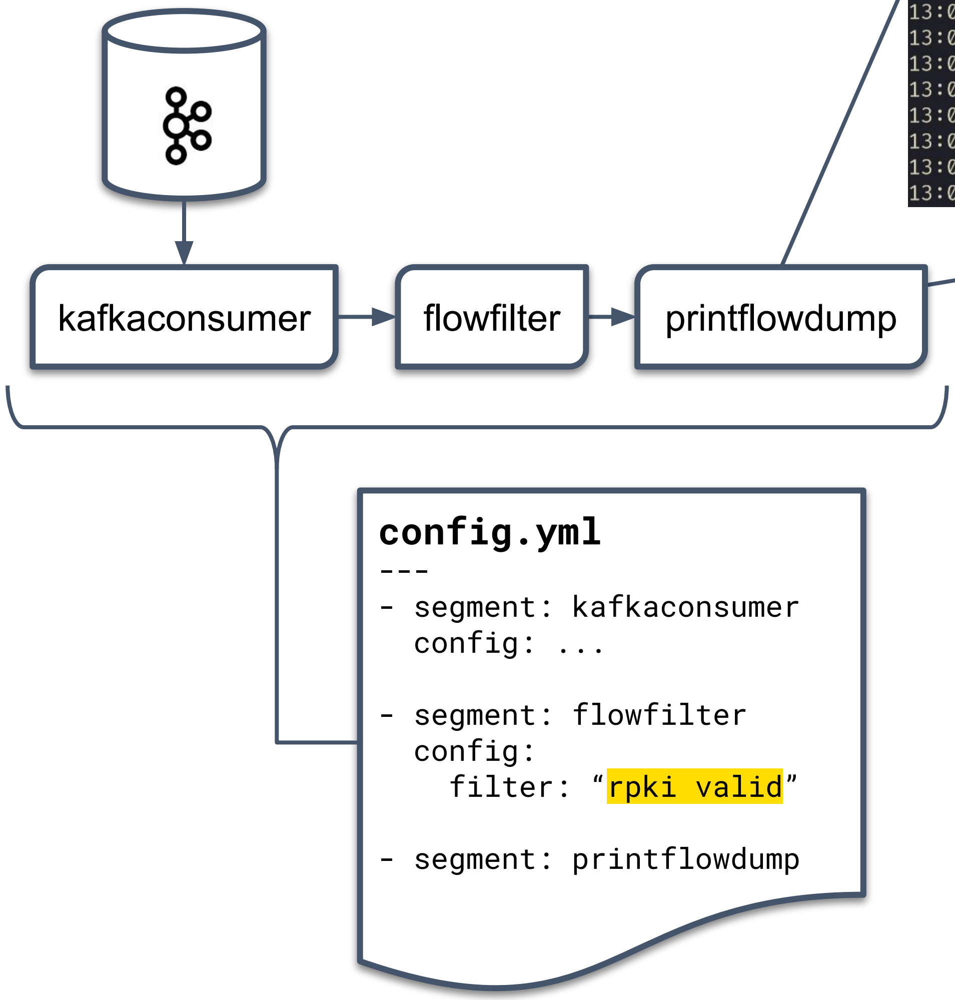
- Built into pipeline to enable high granularity stream processing
- Options include:
  - **prefix tagging**
  - **BGP info**
  - determine remote
  - geolocation
  - DNS
  - SNMP info
  - normalization
  - **anonymization**
  - **filtering** by different means

Enrichment with BGP segment

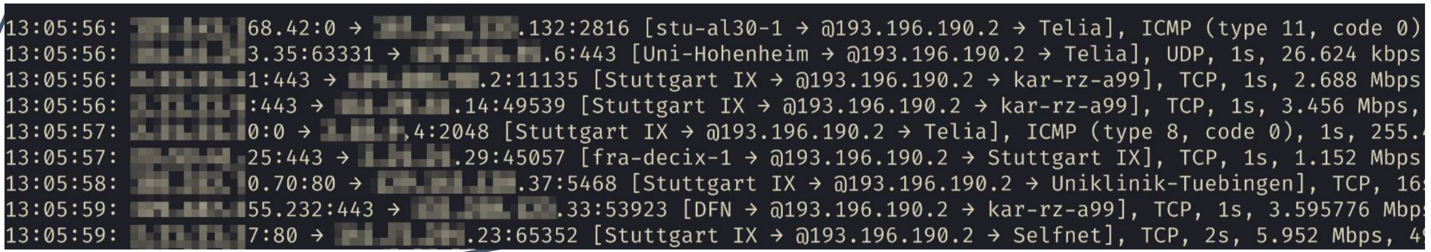


```
config.yml
---
- segment: goflow

- segment: bgp
  config: ...

- segment: kafkaproducer
  config: ...
```

Checking for RPKI (In-)valids

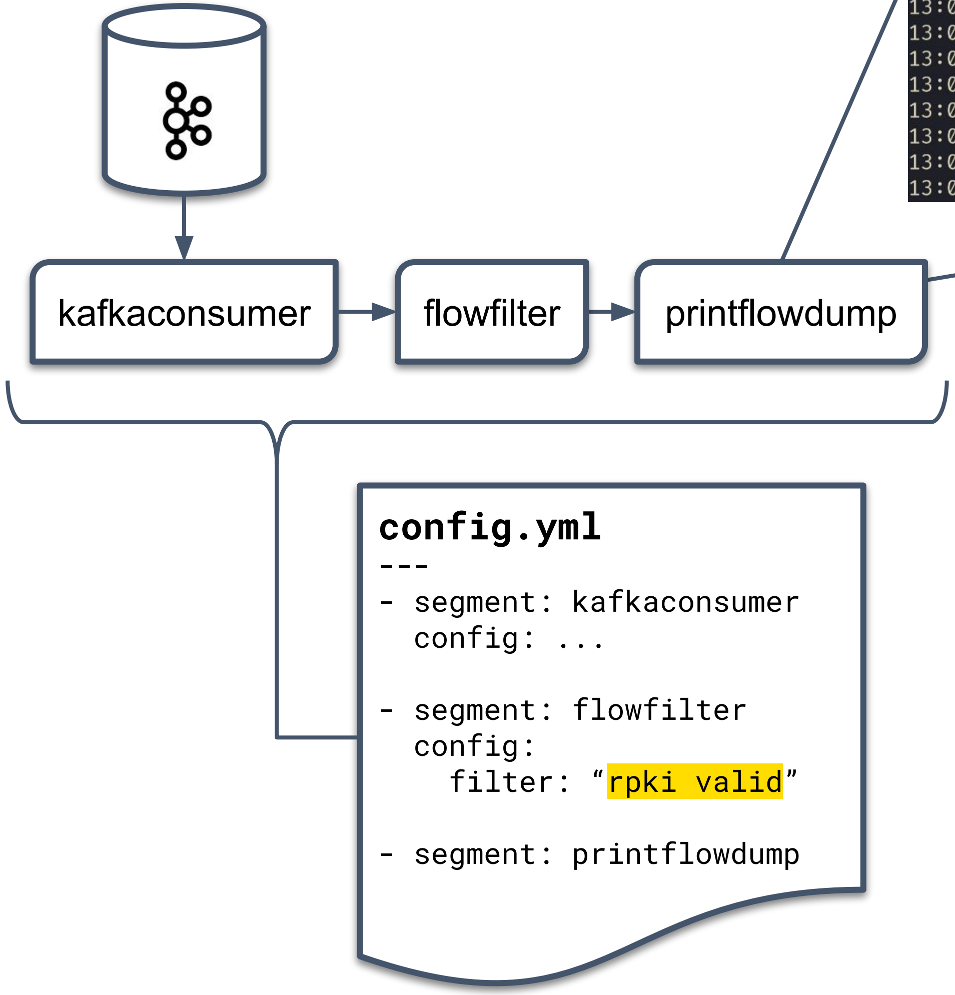


```
config.yml
---
- segment: kafkaconsumer
  config: ...

- segment: flowfilter
  config:
    filter: “rpki valid”

- segment: printflowdump
```

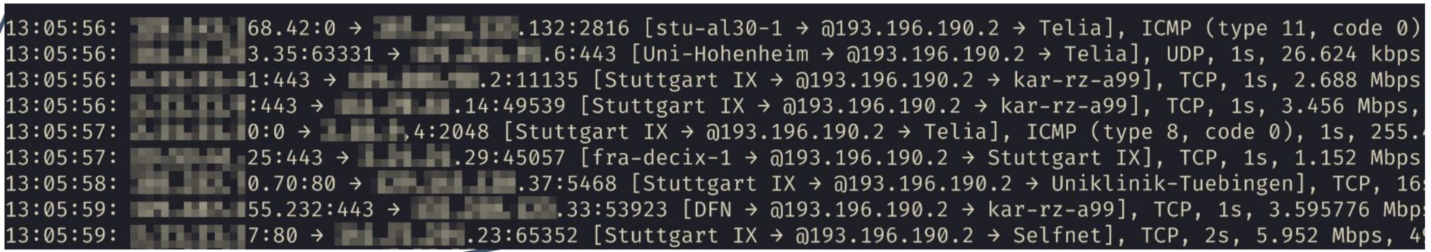


```
13:05:56: [redacted]68.42:0 → [redacted].132:2816 [stu-al30-1 → @193.196.190.2 → Telia], ICMP (type 11, code 0)
13:05:56: [redacted]3.35:63331 → [redacted].6:443 [Uni-Hohenheim → @193.196.190.2 → Telia], UDP, 1s, 26.624 kbps
13:05:56: [redacted]1:443 → [redacted].2:11135 [Stuttgart IX → @193.196.190.2 → kar-rz-a99], TCP, 1s, 2.688 Mbps
13:05:56: [redacted]:443 → [redacted].14:49539 [Stuttgart IX → @193.196.190.2 → kar-rz-a99], TCP, 1s, 3.456 Mbps,
13:05:57: [redacted]0:0 → [redacted].4:2048 [Stuttgart IX → @193.196.190.2 → Telia], ICMP (type 8, code 0), 1s, 255.
13:05:57: [redacted]25:443 → [redacted].29:45057 [fra-decix-1 → @193.196.190.2 → Stuttgart IX], TCP, 1s, 1.152 Mbps
13:05:58: [redacted]0.70:80 → [redacted].37:5468 [Stuttgart IX → @193.196.190.2 → Uniklinik-Tuebingen], TCP, 16
13:05:59: [redacted]55.232:443 → [redacted].33:53923 [DFN → @193.196.190.2 → kar-rz-a99], TCP, 1s, 3.595776 Mbp
13:05:59: [redacted]7:80 → [redacted].23:65352 [Stuttgart IX → @193.196.190.2 → Selfnet], TCP, 2s, 5.952 Mbps, 4
```

Checking for RPKI (In-)valids



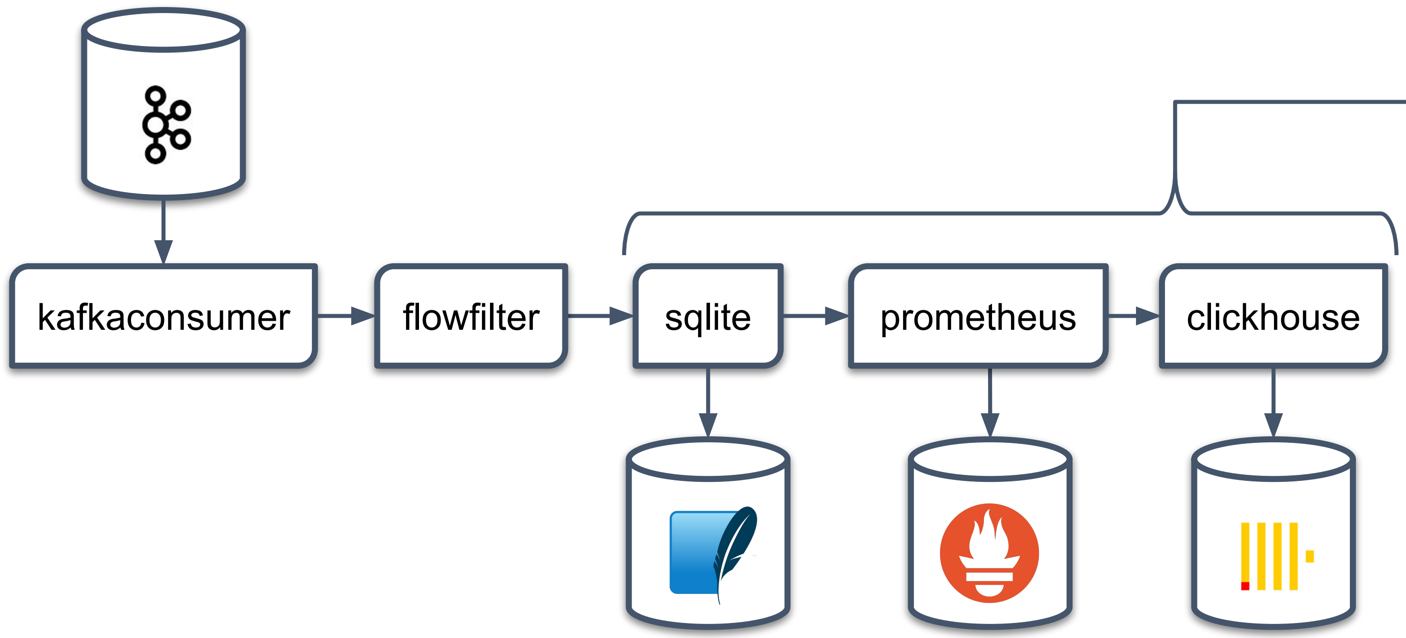
```
config.yml
---
- segment: kafkaconsumer
  config: ...

- segment: flowfilter
  config:
    filter: “rpki valid”

- segment: printflowdump
```



```
13:05:56: [redacted]68.42:0 → [redacted].132:2816 [stu-al30-1 → @193.196.190.2 → Telia], ICMP (type 11, code 0)
13:05:56: [redacted]3.35:63331 → [redacted].6:443 [Uni-Hohenheim → @193.196.190.2 → Telia], UDP, 1s, 26.624 kbps
13:05:56: [redacted]1:443 → [redacted].2:11135 [Stuttgart IX → @193.196.190.2 → kar-rz-a99], TCP, 1s, 2.688 Mbps
13:05:56: [redacted]:443 → [redacted].14:49539 [Stuttgart IX → @193.196.190.2 → kar-rz-a99], TCP, 1s, 3.456 Mbps,
13:05:57: [redacted]0:0 → [redacted].4:2048 [Stuttgart IX → @193.196.190.2 → Telia], ICMP (type 8, code 0), 1s, 255.
13:05:57: [redacted]25:443 → [redacted].29:45057 [fra-decix-1 → @193.196.190.2 → Stuttgart IX], TCP, 1s, 1.152 Mbps
13:05:58: [redacted]0.70:80 → [redacted].37:5468 [Stuttgart IX → @193.196.190.2 → Uniklinik-Tuebingen], TCP, 16
13:05:59: [redacted]55.232:443 → [redacted].33:53923 [DFN → @193.196.190.2 → kar-rz-a99], TCP, 1s, 3.595776 Mbp
13:05:59: [redacted]7:80 → [redacted].23:65352 [Stuttgart IX → @193.196.190.2 → Selfnet], TCP, 2s, 5.952 Mbps, 4
```

- Other filters: `rpki invalid`, `passes-through 559`, `localpref >100`
- Use an additional statistical filter segment to limit to elephant flows

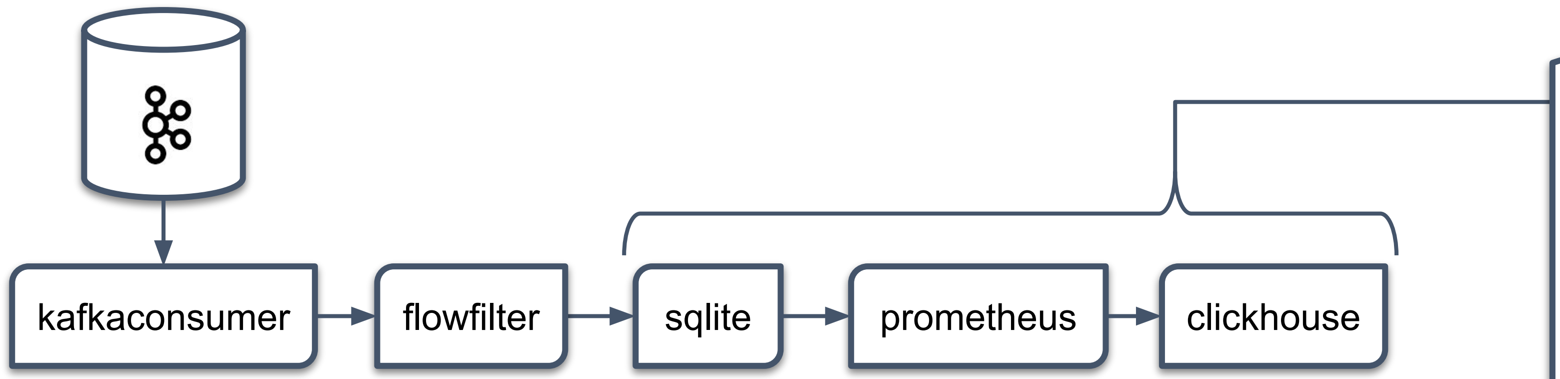
Doing something useful with those Flows...

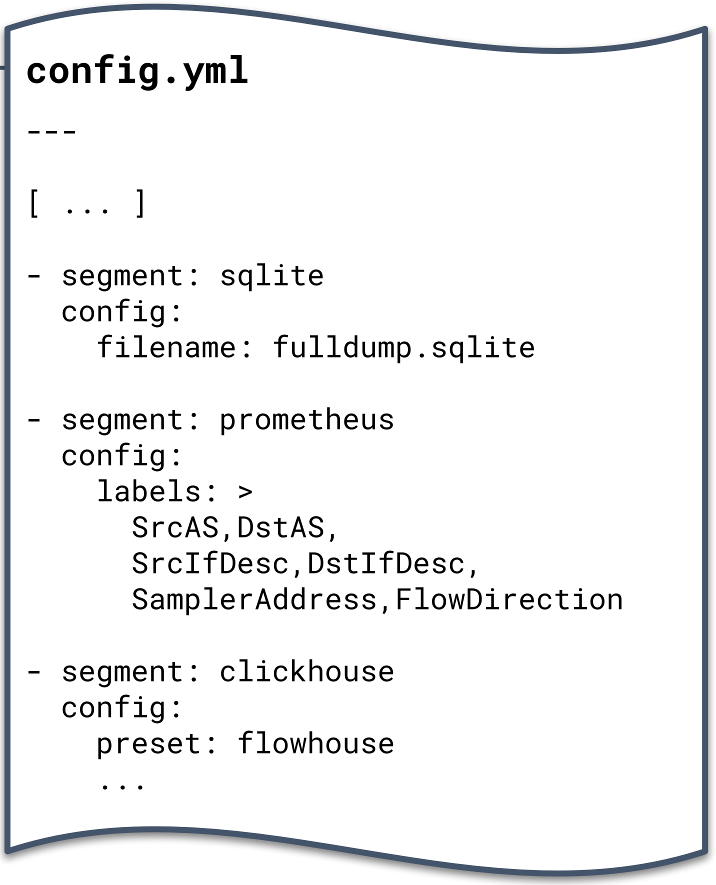


config.yml

```
---

[ ... ]

- segment: sqlite
  config:
    filename: fulldump.sqlite

- segment: prometheus
  config:
    labels: >
      SrcAS,DstAS,
      SrcIfDesc,DstIfDesc,
      SamplerAddress,FlowDirection

- segment: clickhouse
  config:
    preset: flowhouse
    ...
```

Doing something useful with those Flows...

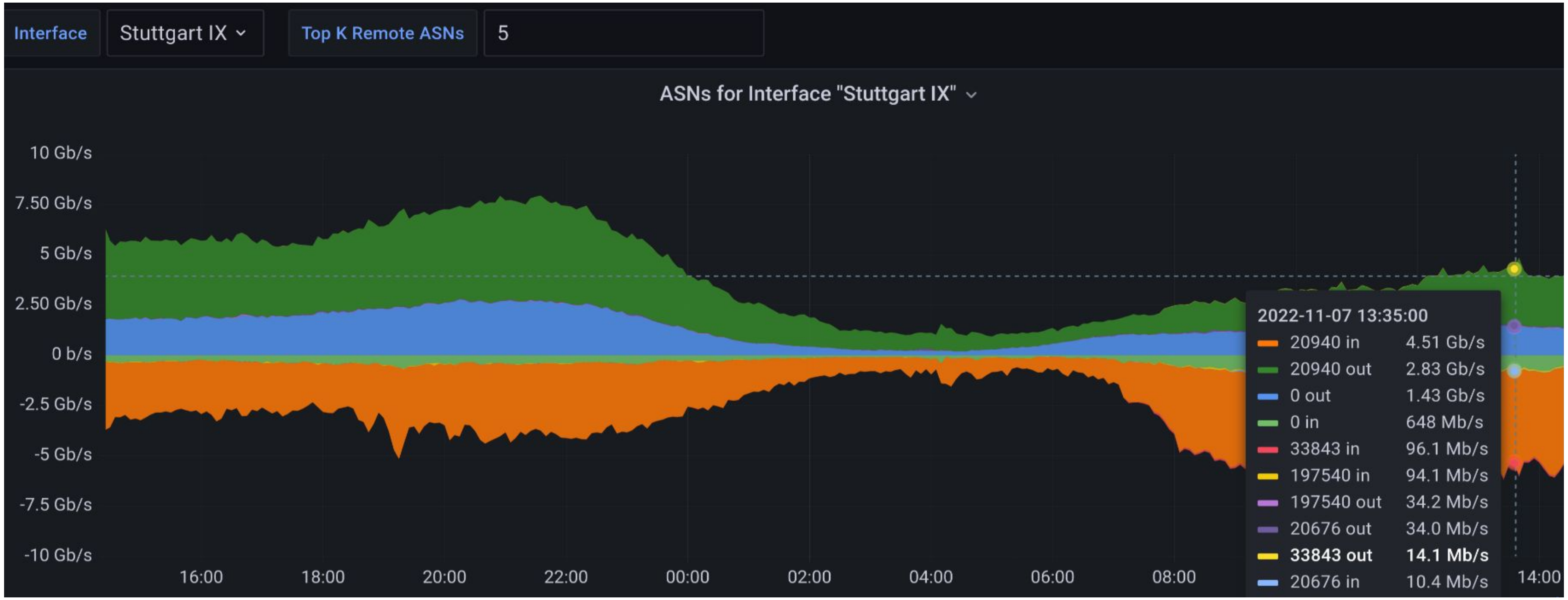




```
config.yml
---

[ ... ]

- segment: sqlite
  config:
    filename: fulldump.sqlite

- segment: prometheus
  config:
    labels: >
      SrcAS,DstAS,
      SrcIfDesc,DstIfDesc,
      SamplerAddress,FlowDirection

- segment: clickhouse
  config:
    preset: flowhouse
    ...
```

- All segments – inputs, outputs, or otherwise – chain into each other
- Multiple instances of one segment to export different dimensions of flows

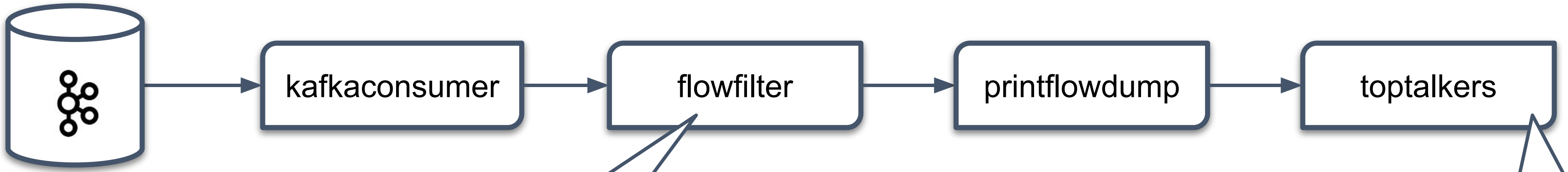
Finding Candidates for private Peering



2022-11-07 13:35:00

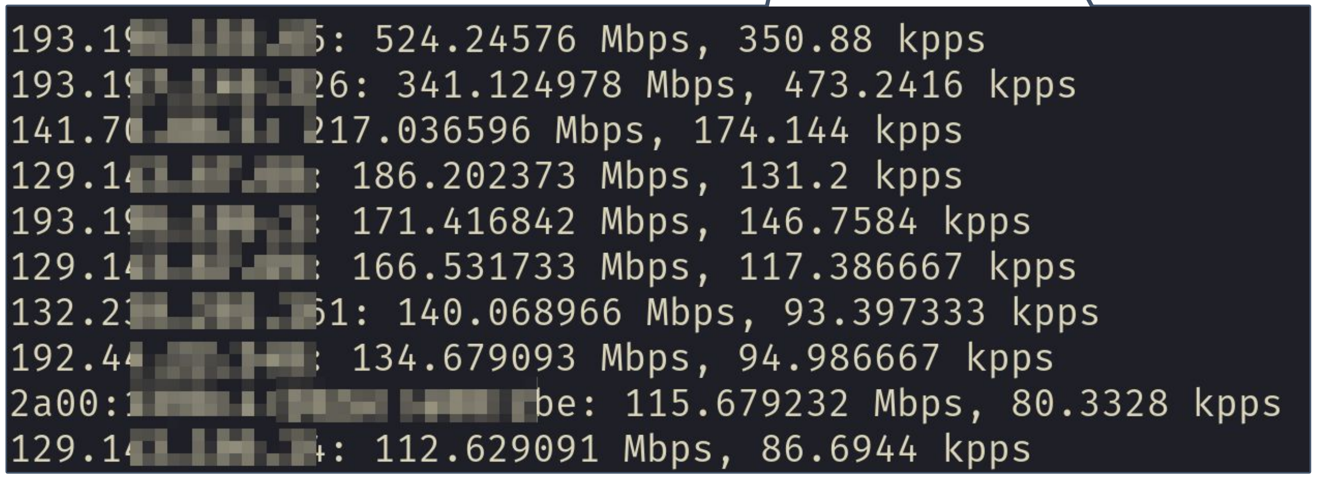
🟧 20940 in	4.51 Gb/s
🟩 20940 out	2.83 Gb/s
🟦 0 out	1.43 Gb/s
🟩 0 in	648 Mb/s
🟥 33843 in	96.1 Mb/s
🟨 197540 in	94.1 Mb/s
🟪 197540 out	34.2 Mb/s
🟪 20676 out	34.0 Mb/s
🟨 **33843 out**	**14.1 Mb/s**
🟦 20676 in	10.4 Mb/s

Determine DoS Recipient Addresses



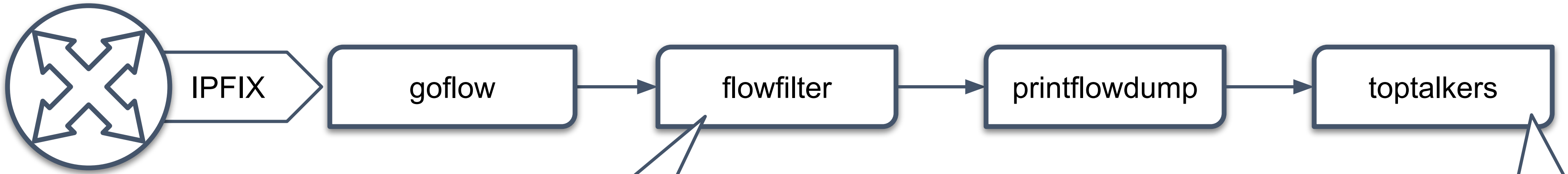
1. Start filtering for what seems to be causing problems:
   `proto udp and src port 123`

2. Include fragmented datagrams:
   `... and port 0`



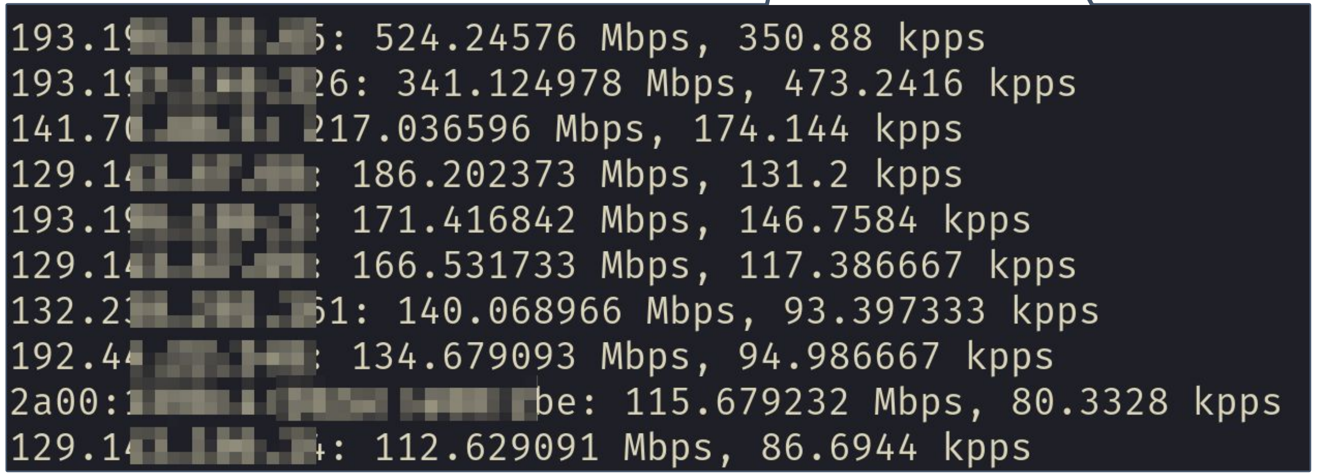
```
193.1[redacted]5: 524.24576 Mbps, 350.88 kpps
193.1[redacted]26: 341.124978 Mbps, 473.2416 kpps
141.7[redacted]217.036596 Mbps, 174.144 kpps
129.1[redacted]: 186.202373 Mbps, 131.2 kpps
193.1[redacted]: 171.416842 Mbps, 146.7584 kpps
129.1[redacted]: 166.531733 Mbps, 117.386667 kpps
132.2[redacted]61: 140.068966 Mbps, 93.397333 kpps
192.4[redacted]: 134.679093 Mbps, 94.986667 kpps
2a00:[redacted]be: 115.679232 Mbps, 80.3328 kpps
129.1[redacted]4: 112.629091 Mbps, 86.6944 kpps
```

Determine DoS Recipient Addresses



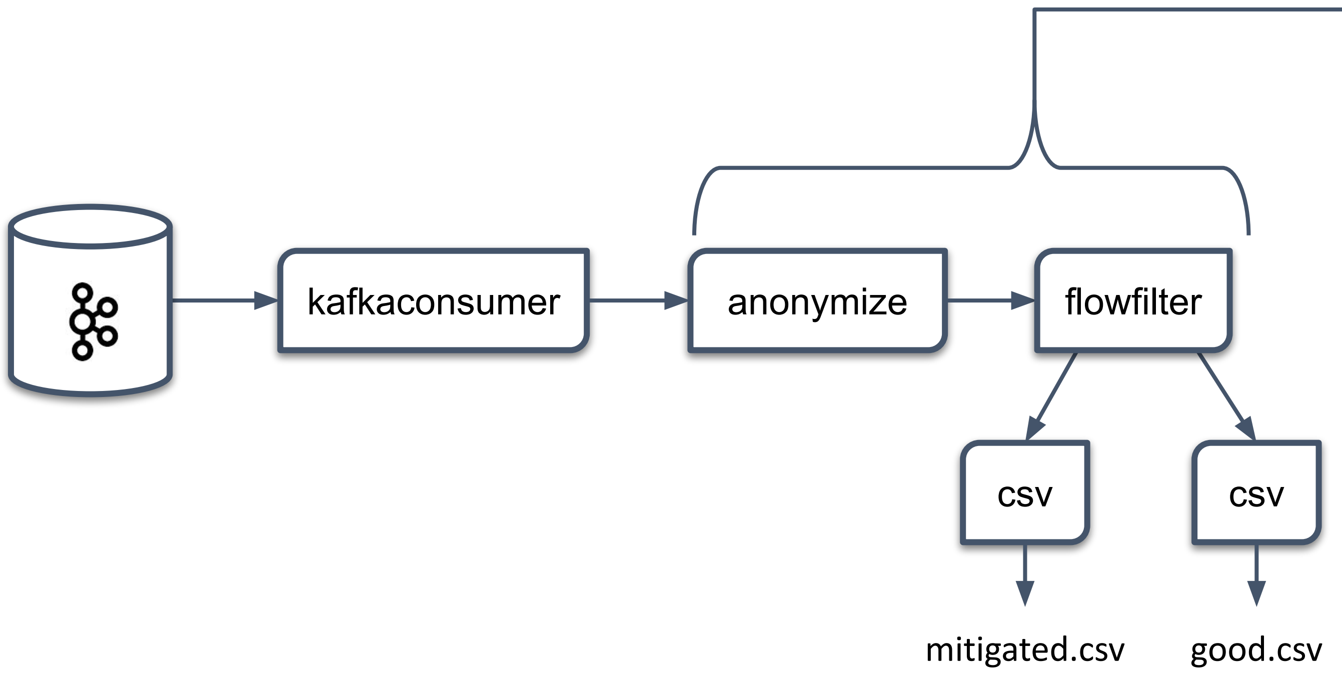
1. Limit results to SYN flooding
   `tcpflags syn and not tcpflags ack and pps >9000`

2. Check whether mitigations are working:
   `... and not status dropped`



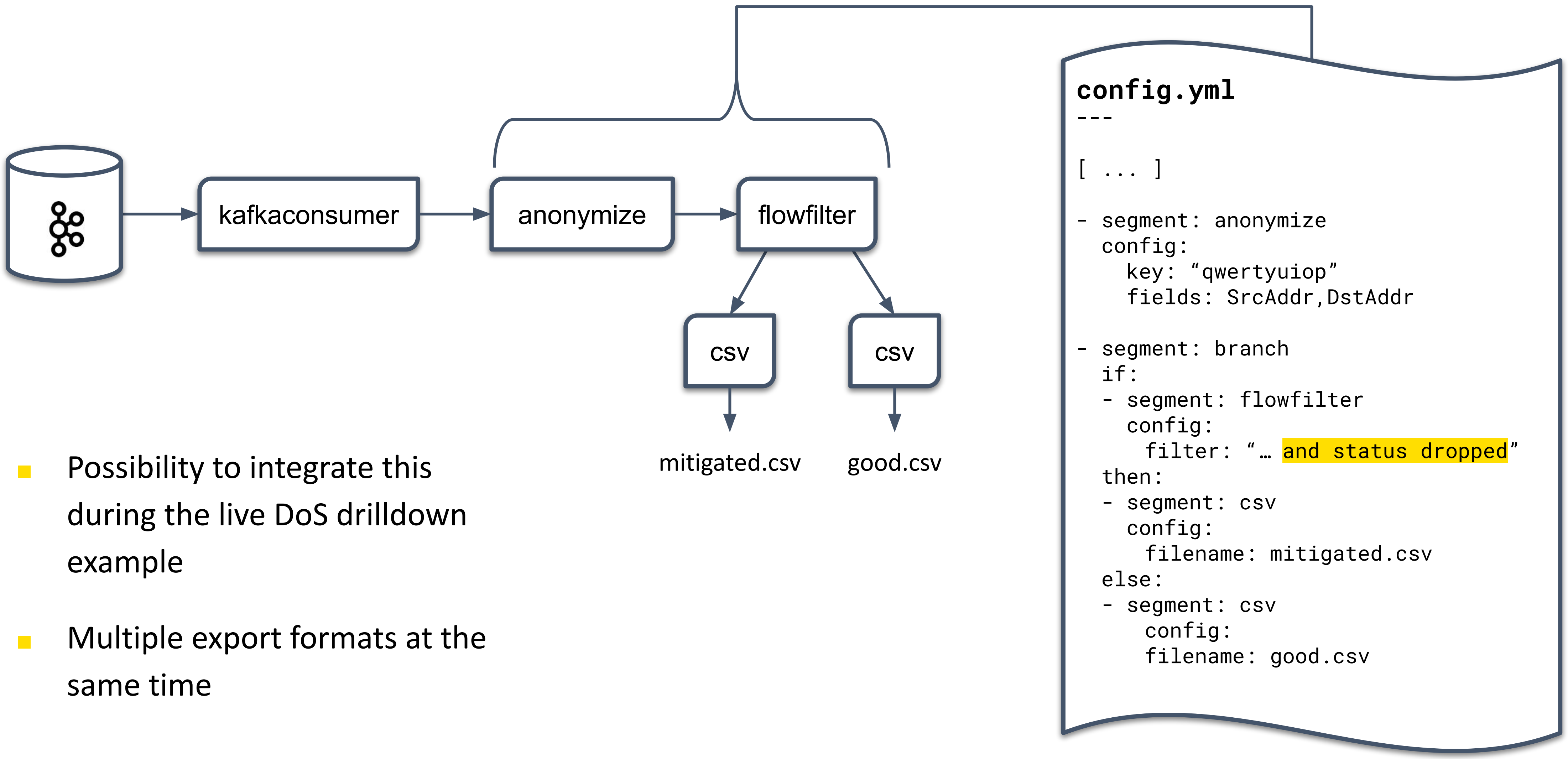
```
193.1[redacted]5: 524.24576 Mbps, 350.88 kpps
193.1[redacted]26: 341.124978 Mbps, 473.2416 kpps
141.7[redacted]217.036596 Mbps, 174.144 kpps
129.1[redacted]: 186.202373 Mbps, 131.2 kpps
193.1[redacted]: 171.416842 Mbps, 146.7584 kpps
129.1[redacted]: 166.531733 Mbps, 117.386667 kpps
132.2[redacted]51: 140.068966 Mbps, 93.397333 kpps
192.4[redacted]: 134.679093 Mbps, 94.986667 kpps
2a00:[redacted]be: 115.679232 Mbps, 80.3328 kpps
129.1[redacted]4: 112.629091 Mbps, 86.6944 kpps
```

Labeled Datasets for Researchers



```
config.yml
---

[ ... ]

- segment: anonymize
  config:
    key: “qwertyuiop”
    fields: SrcAddr,DstAddr

- segment: branch
  if:
  - segment: flowfilter
    config:
      filter: “… and status dropped”
  then:
  - segment: csv
    config:
      filename: mitigated.csv
  else:
  - segment: csv
      config:
      filename: good.csv
```

Labeled Datasets for Researchers



```
config.yml
---

[ ... ]

- segment: anonymize
  config:
    key: “qwertyuiop”
    fields: SrcAddr,DstAddr

- segment: branch
  if:
  - segment: flowfilter
    config:
      filter: “… and status dropped”
  then:
  - segment: csv
    config:
      filename: mitigated.csv
  else:
  - segment: csv
      config:
      filename: good.csv
```

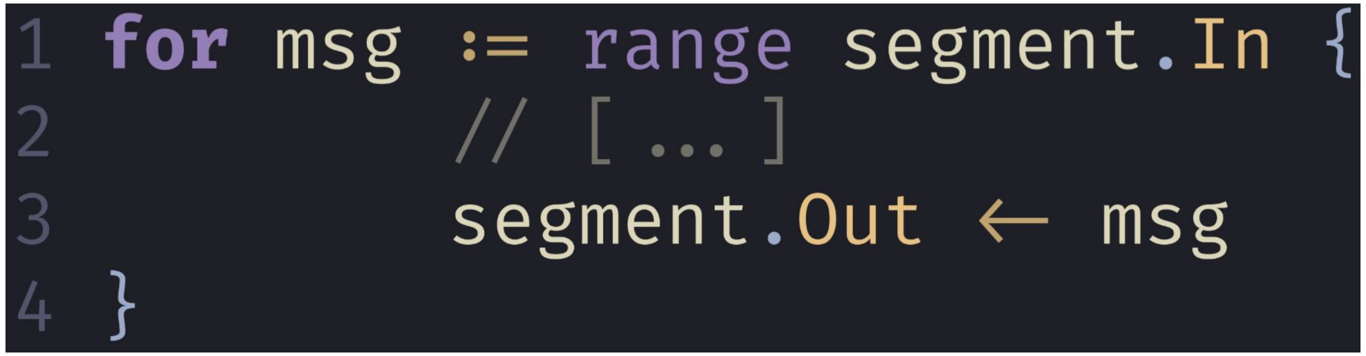
- Possibility to integrate this during the live DoS drilldown example
- Multiple export formats at the same time

Extensibility and Integration

- Flowpipelines are extensible using custom segments, powered by the Go plugin system
- Allows implementation of **any** algorithm, data structure, or filtering
- Pluggable into existing configurations

Custom Segments

1. Code in the provided template (or copy a stock segment)
2. Compile it: `go build -buildmode=plugin ./custom.go`
3. Use the assigned name in a config and launch with `-p custom.so`



```go
for msg := range segment.In {
        // [...]
        segment.Out ← msg
}
```

Thank you for your attention!

Questions?

https://github.com/bwNetFlow/flowpipeline

Daniel Nägele – naegele@belwue.de – @debugloop (on IRC & social)