KEYSIGHT

SAI Challenger Enhancements for DASH Testing Episode II

Chris Sommers, Mircea Dan Georghe - Keysight
Anton Putria, PLVision
2022-10-12

Agenda

- Recap from last meeting – Why, what, when
- Outbound routing VNET test scaling approach using snappi and SAI-Challenger
- Setups overview
- Redis test on saivs
- Next Steps
- Call to Action

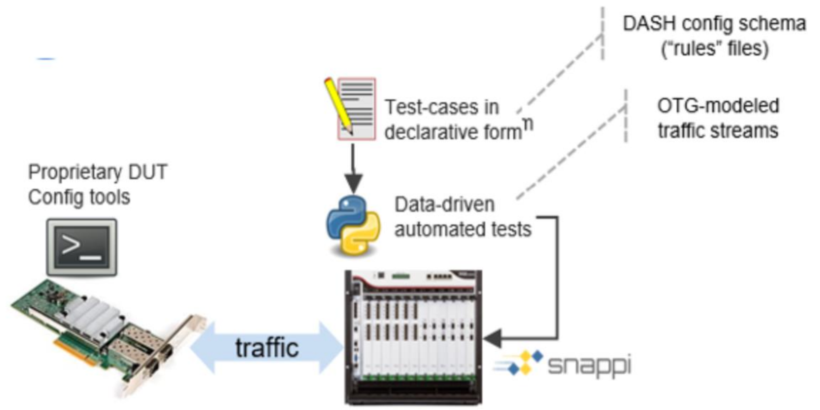
Why, What and When?

Why? *DASH stretches the limits of traditional testing!*

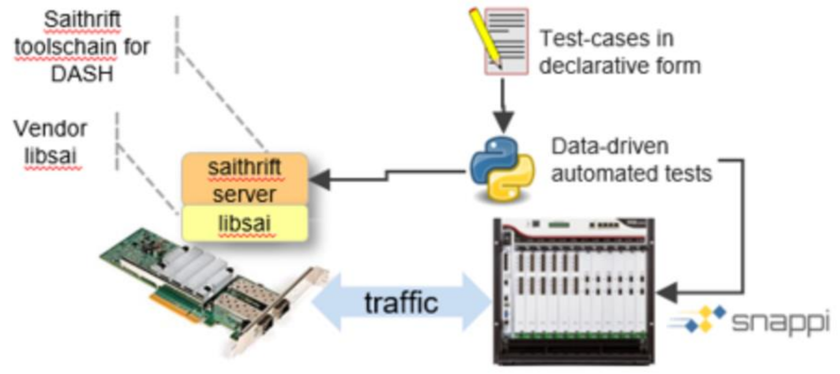
- Complex test-cases - many tables & interdependencies
- Huge table scale (millions of entries)
- Multiple APIs to test: SAI, sairedis, gNMI
- Performance testing of HW Targets (line rate)
- SW devs are increasingly expected to write test cases – how to make it easier?

Recap – DASH Test Maturity Stages

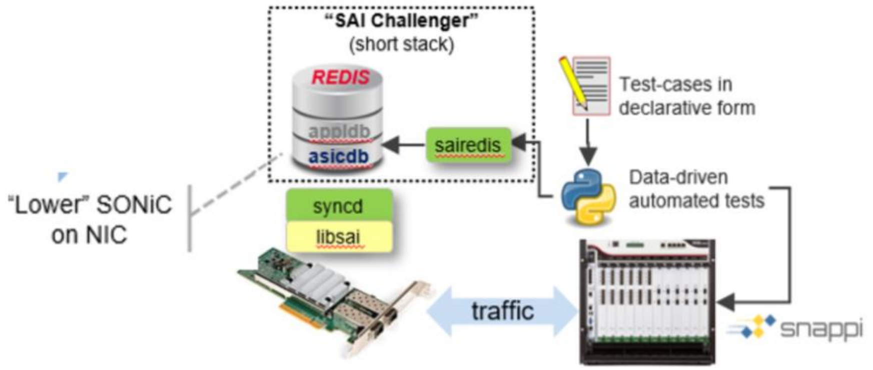
Data plane Testing Stage 2: Standardized, Automated Test Cases


DASH config schema ("rules" files)
Test-cases in declarative form
OTG-modeled traffic streams
Proprietary DUT Config tools
Data-driven automated tests
traffic
snappi

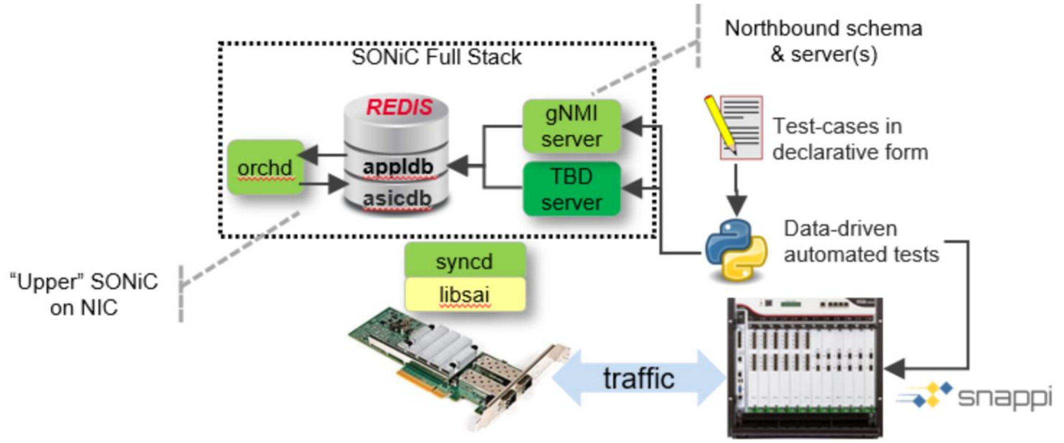
Data plane Testing Stage 3: DUT configuration via SAI-Thrift


Saithrift toolschain for DASH
Test-cases in declarative form
Vendor libsai
saithrift server
libsai
Data-driven automated tests
traffic
snappi

Data plane Testing Stage 4: DUT configuration via SAI-Redis


"SAI Challenger" (short stack)
REDIS
appldb
asicdb
sairedis
Test-cases in declarative form
"Lower" SONiC on NIC
syncd
libsai
Data-driven automated tests
traffic
snappi

Data plane Testing Stage 5: DUT configuration via SONiC Northbound API



KEYSIGHT

https://github.com/Azure/DASH/blob/main/test/docs/dash-test-maturity-stages.md

Why, What and When?

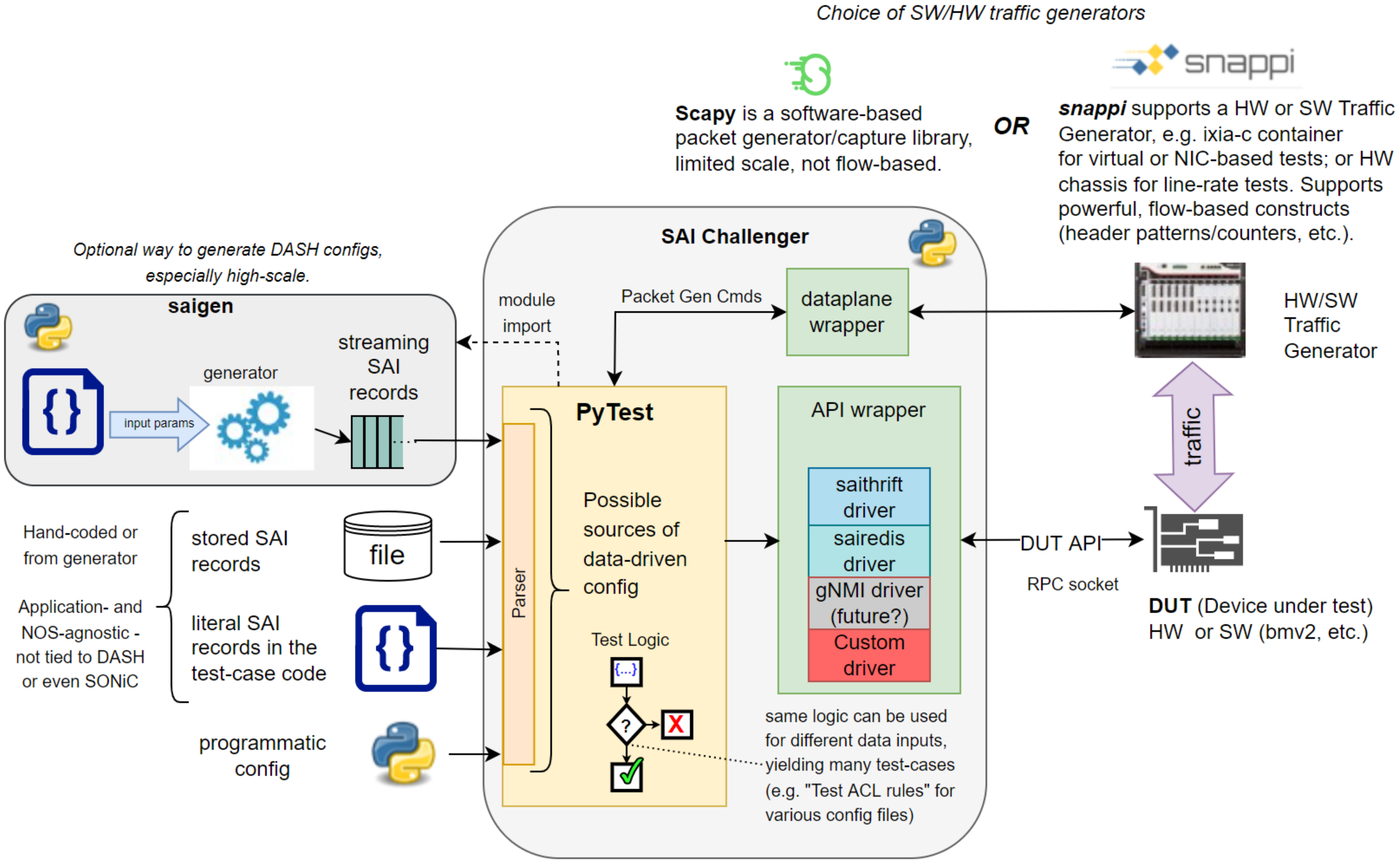
What? *GitHub contributions to SAI and DASH*

- OCP **SAI-Challenger** with Keysight-sponsored enhancements (for *any* SAI device)
- Keysight DASH Config generator can feed test cases for large-scale tests
- Increased developer productivity – focus on declarative configuration **data** and **test logic**, not API plumbing!
- Enhancements for multi-APIs, flexible traffic generators (SW or HW)

When? *Now!*

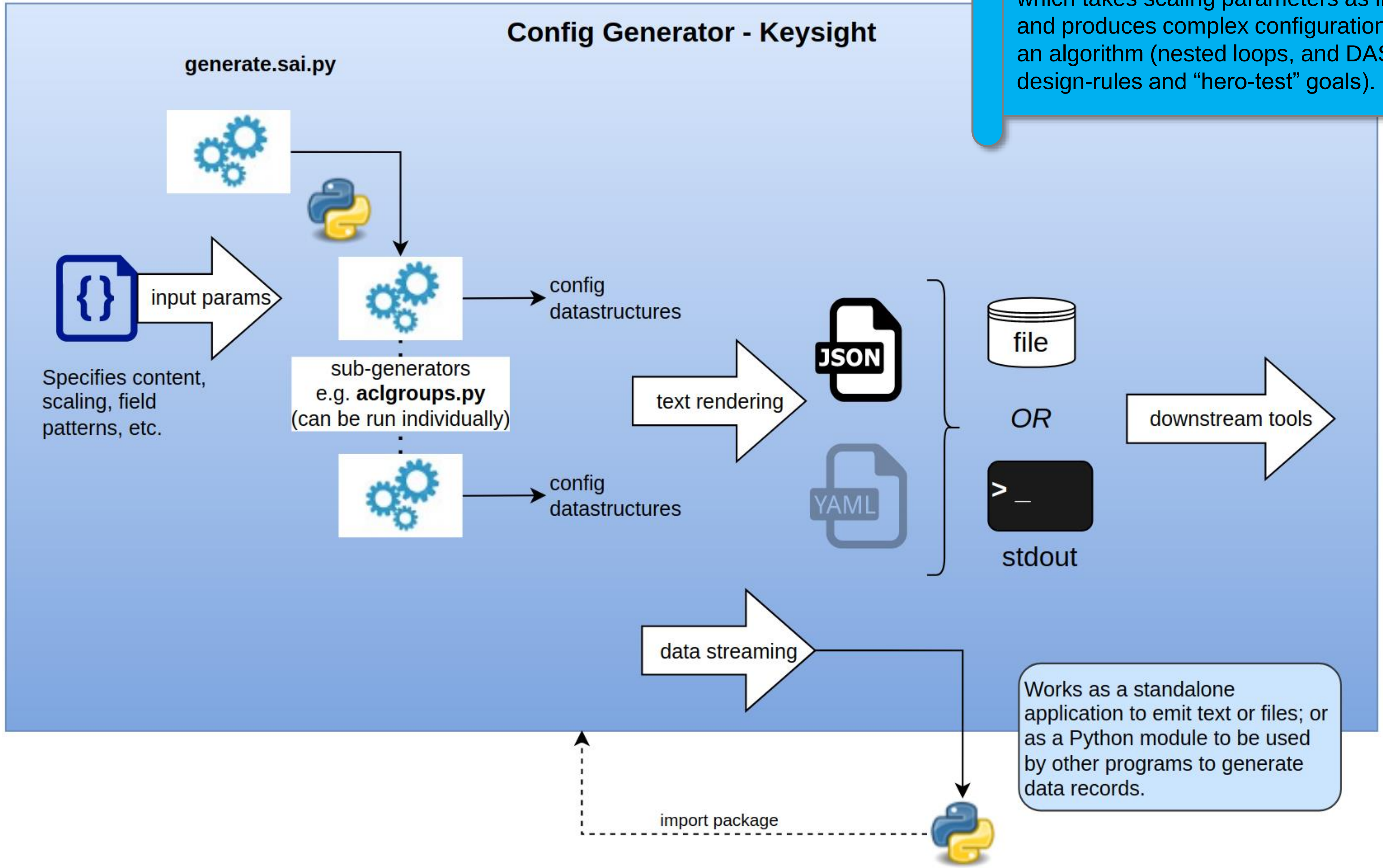
- “Pre-release” Demo today for community
- First “release” pull-request ~ Oct 14

Framework at a Glance



Choice of SW/HW traffic generators
snappi
Scapy is a software-based packet generator/capture library, limited scale, not flow-based.
OR
snappi supports a HW or SW Traffic Generator, e.g. ixia-c container for virtual or NIC-based tests; or HW chassis for line-rate tests. Supports powerful, flow-based constructs (header patterns/counters, etc.).
Optional way to generate DASH configs, especially high-scale.
saigen
input params
generator
streaming SAI records
module import
SAI Challenger
Packet Gen Cmds
dataplane wrapper
HW/SW Traffic Generator
traffic
PyTest
Parser
Possible sources of data-driven config
Test Logic
{...}
?
X
API wrapper
saithrift driver
sairedis driver
gNMI driver (future?)
Custom driver
DUT API
RPC socket
DUT (Device under test) HW or SW (bmv2, etc.)
Hand-coded or from generator
stored SAI records
file
Application- and NOS-agnostic - not tied to DASH or even SONiC
literal SAI records in the test-case code
programmatic config
same logic can be used for different data inputs, yielding many test-cases (e.g. "Test ACL rules" for various config files)

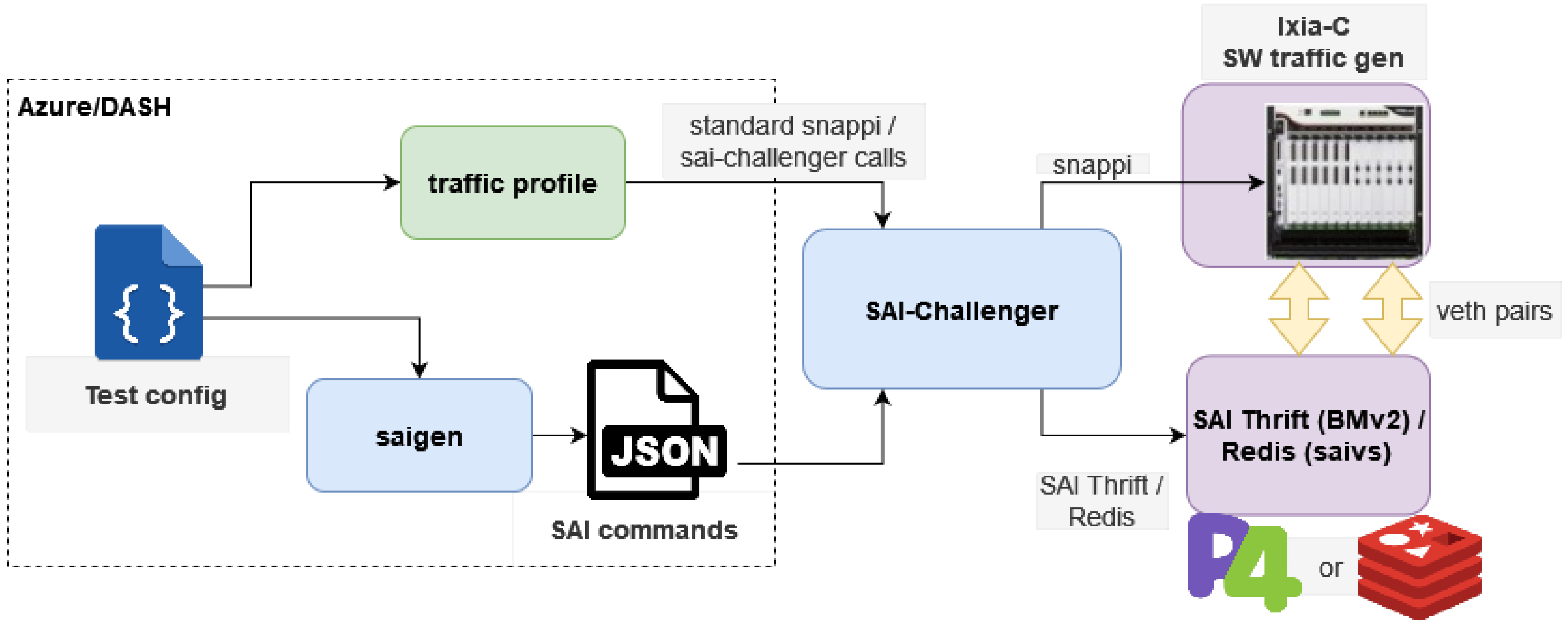
saigen (generator) - Recap



The config generator is like a "wizard" which takes scaling parameters as inputs and produces complex configurations using an algorithm (nested loops, and DASH design-rules and "hero-test" goals).

Works as a standalone application to emit text or files; or as a Python module to be used by other programs to generate data records.

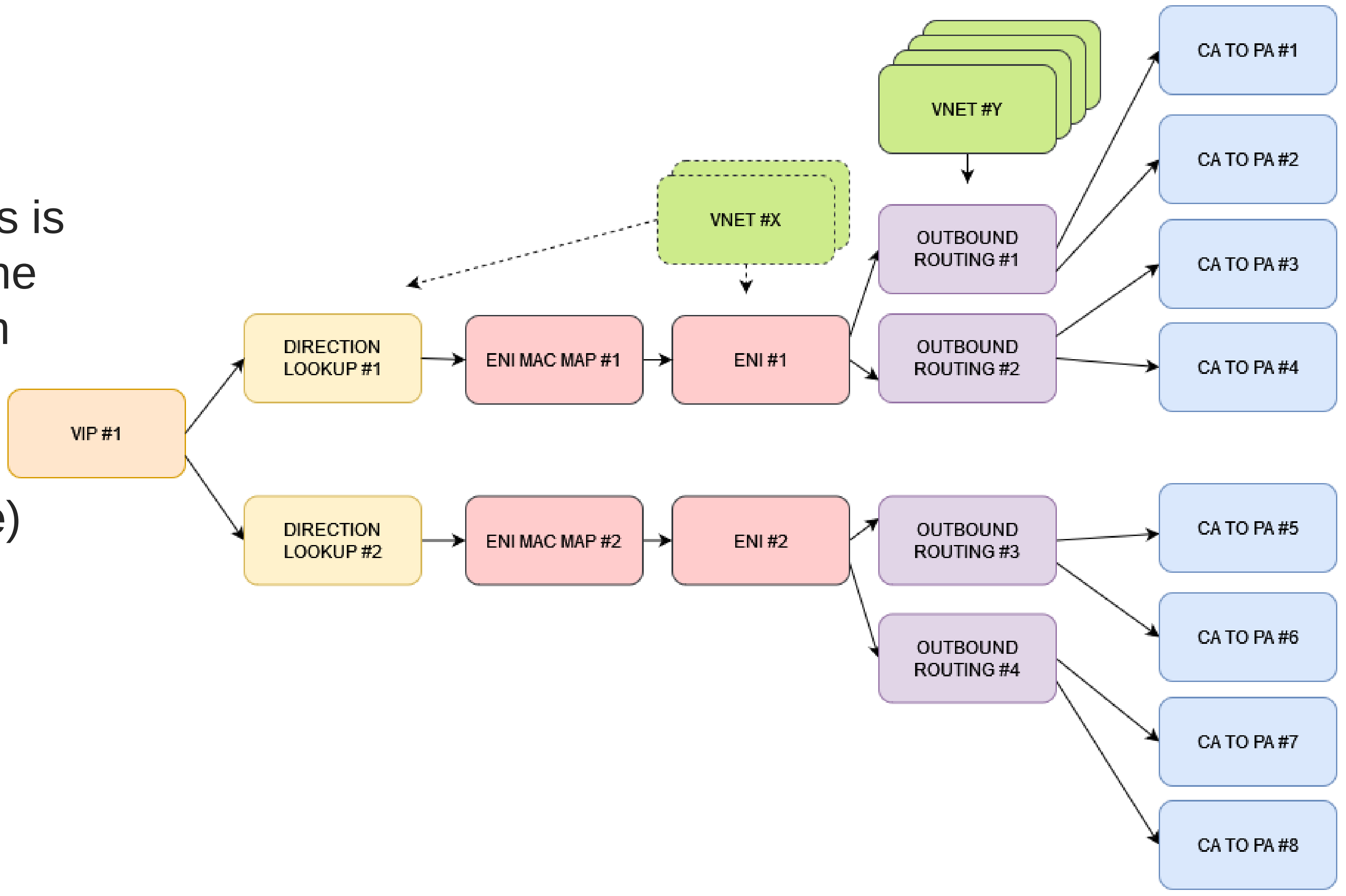
Components overview



- **Test config** – The single source of Truth. Written in the DASH config style.
- **saigen** – scalable SAI config generator
- **traffic profile** – scalable traffic configurator

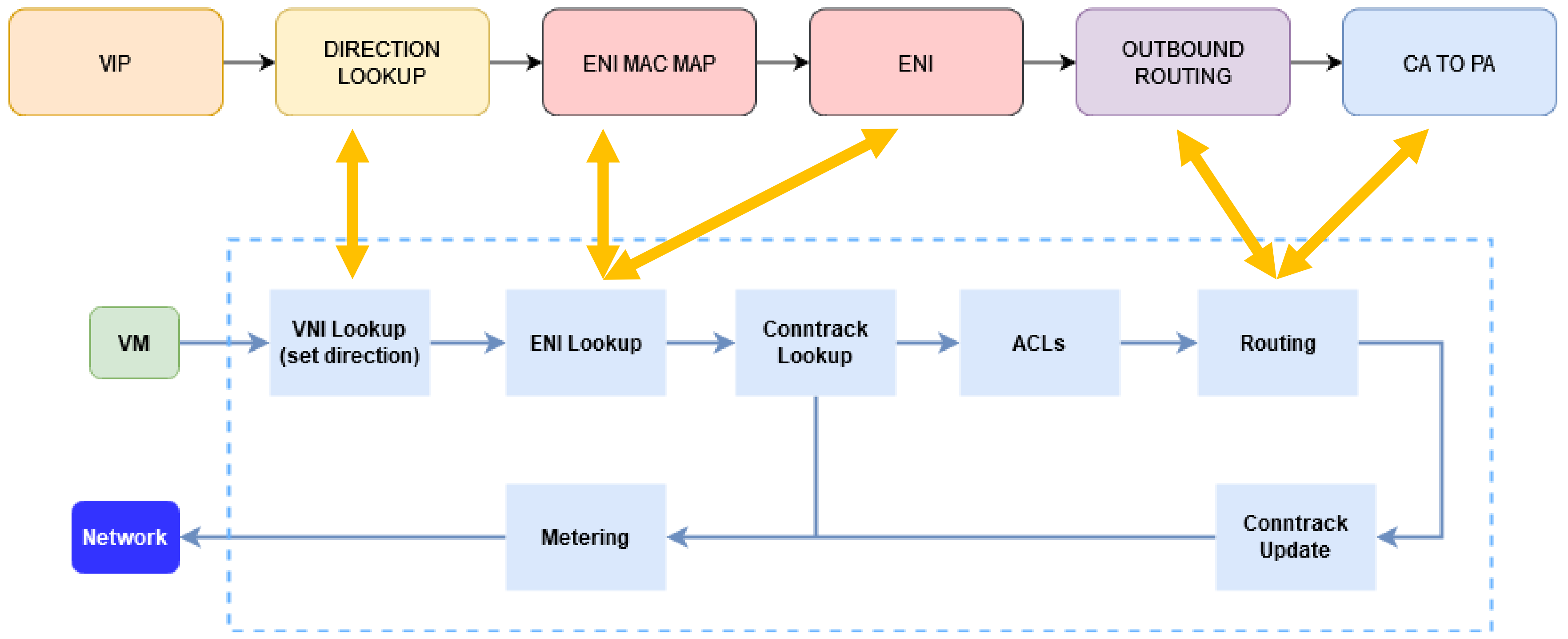
DASH scale configuration (demo)

- The numbers of the entities is autogenerated based on the test config and affects both DASH and traffic configurations.
- (**saigen** and **traffic profile**)



KEYSIGHT

DASH scale configuration (explained)



Scaling configuration

```
'DASH_VNET': {
    'vnet': {
        'VNI': {
            'count’: <NUMBER_OF_VNETs>,
            'start': 1000,
            'step': 1
        }
    }
 }
```

```
'DASH_ENI_ETHER_ADDRESS_MAP': {
    'eam': {
        'count’: <NUMBER_OF_EAM>,
        'SWITCH_ID': '$SWITCH_ID',
        'MAC': {
            'count’: <NUMBER_OF_EAM>,
            'start': '00:CC:CC:CC:00:00',
            'step': "00:00:00:00:00:01"
        },
        'ENI_ID': {
            'count’: <NUMBER_OF_ENI>,
            'start’: $eni_#{0}'
        }
    }
 }
```

DASH high-level config vs. sai-thrift API calls

“DASH Config” format - Abstracted

```
'DASH_OUTBOUND_ROUTING': {
    'ore': {
        'SWITCH_ID': '$SWITCH_ID',
        'ENI_ID': '$eni_#1',
        'DESTINATION': "10.1.2.0/24",
        'ACTION': 'ROUTE_VNET',
        'DST_VNET_ID': '$vnet_#1'
    }
}
```

Traditional PTF: sai-thrift direct API calls

```
ca_prefix_1 = sai_thrift_ip_prefix_t(addr_family=SAI_IP_ADDR_FAMILY_IPV4,
                                     addr=sai_thrift_ip_addr_t(ip4="10.1.2.0"),
                                     mask=sai_thrift_ip_addr_t(ip4="255.255.255.0"))
ore_1 = sai_thrift_outbound_routing_entry_t(switch_id=switch_id, eni_id=eni_1, destination=ca_prefix_1)
status = sai_thrift_create_outbound_routing_entry(self.client, ore_1,
                                                  action=SAI_OUTBOUND_ROUTING_ENTRY_ACTION_ROUTE_VNET,
                                                  dst_vnet_id=vnet_1)
```

Scaling configuration -> SAI JSON format

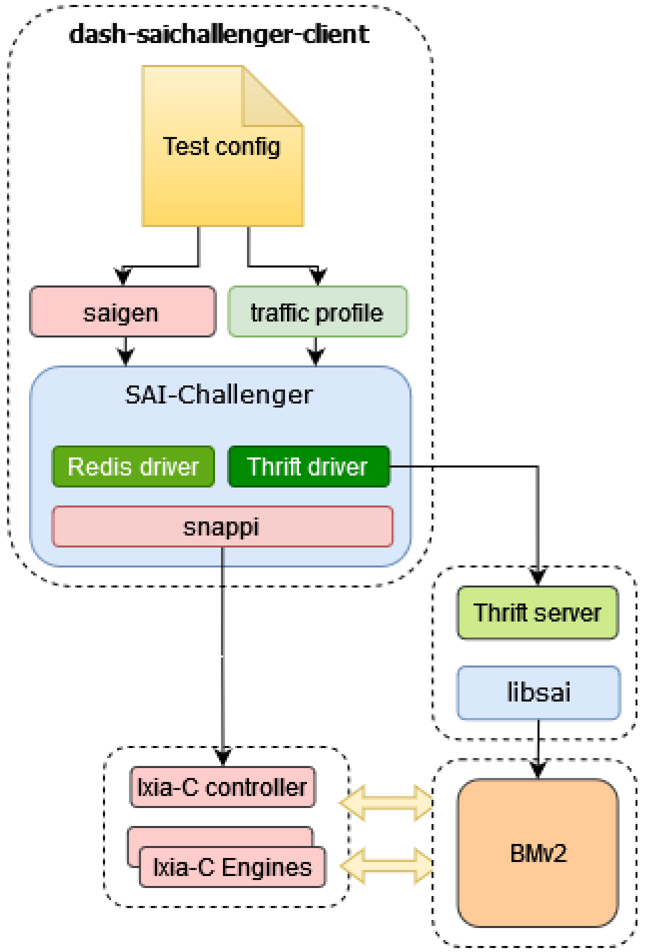
Input to generator and traffic profile

```
'DASH_OUTBOUND_ROUTING': {
    'ore': {
        'SWITCH_ID': '$SWITCH_ID',
        'ENI_ID': '$eni_#1',
        'DESTINATION': "10.1.2.0/24",
        'ACTION': 'ROUTE_VNET',
        'DST_VNET_ID': '$vnet_#1'
    }
}
```

```
{
  "name": "ore_#1",
  "op": "create",
  "type": "SAI_OBJECT_TYPE_OUTBOUND_ROUTING_ENTRY",
  "key": {
    "switch_id": "$SWITCH_ID",
    "eni_id": "$eni_#1",
    "destination": "10.1.2.0/24"
  },
  "attributes": [
    "SAI_OUTBOUND_ROUTING_ENTRY_ATTR_ACTION",
    "SAI_OUTBOUND_ROUTING_ENTRY_ACTION_ROUTE_VNET",
    "SAI_OUTBOUND_ROUTING_ENTRY_ATTR_DST_VNET_ID",
    "$vnet_#1"
  ]
}
```

Demo Time!

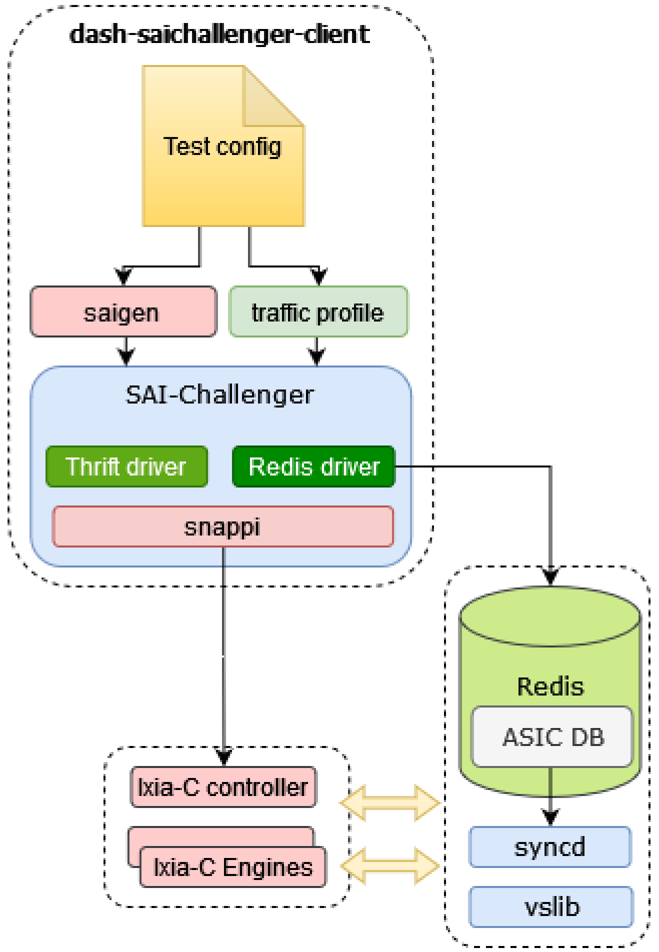
Setups overview



Same test framework for both:

- SAI Thrift
- Redis

Same SAI based DUT configuration



Why SAI-Redis testing

1. Simplify syncd development using SONiC-lite (short stack: only syncd and redis) - no need to build the whole SONiC infrastructure, make and run procedures are much faster.
2. Ensure proper syncd operation and linking to libsai and Redis – environment is very close to “real” SONiC.
3. Replay real world use case by writing commands directly to ASIC_DB – no need to run multiple dockers.
4. Possibility to create own use case using SAI (records in Redis ASIC_DB).
5. Easy to scale – string-based Redis API allows generating multiple commands in a simple way.
6. *Proven* by experience to accelerate and simplify syncd development & SONiC integration

SAI-Challenger Advantages - Roundup

1. Abstraction level for using multiple types of DUTs (NPU, DPU)
2. Abstraction level for using multiple types of Dataplanes (PTF, snappi)
3. Pytest based – unlocks the whole ecosystem with multiple plugins and integrations
4. SONiC-lite stack – simplifies syncd development
5. Dockerized environment
6. Testbed agnostic test cases – all testbed-dependent configuration defined by a JSON file.
7. Multi-DUT support – multiple DUTs in a single testbed.
8. OCP official project

Added in scope of this demo

1. Multiple DUT APIs support (SAI-Thrift, Redis). Allows to add new custom drivers.
2. Snappi support.

Next steps

Immediate:

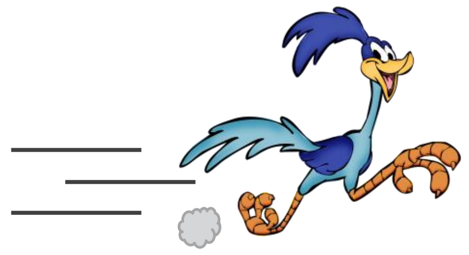
- Finishing touches, pull-requests, merge to opencompute/SAI and Azure/DASH

RoadMap:

- More test cases – pending bmv2 progress (IPv6, vnet_in, ACLs, …)
  - Many new test-cases will fail (feature broken/not ready) – Pytest @mark.xfail until fixed
  - Issues will be filed against bmv2
- Test on real DPUs – pending vendor sai/saithrift implementation
  - *In the meantime*…if vendors agree, we can publish proprietary configs & generators which work in the lab on hardware
- SAI-Redis tests using SAME test cases– pending vendor implementation of syncd, redis
- gNMI enhancements, test-cases – depending upon community interest. Use same configs as SAI, SAI-redis to verify the whole stack a layer at a time

Community Call to Action

- Try out the new framework, give feedback
- Fix/complete basic Bmv2 VNET features
- Finish Bmv2 stateful behavior
- Vendors – implement sai_thrift on your DPUs so we can test @ speed & scale!



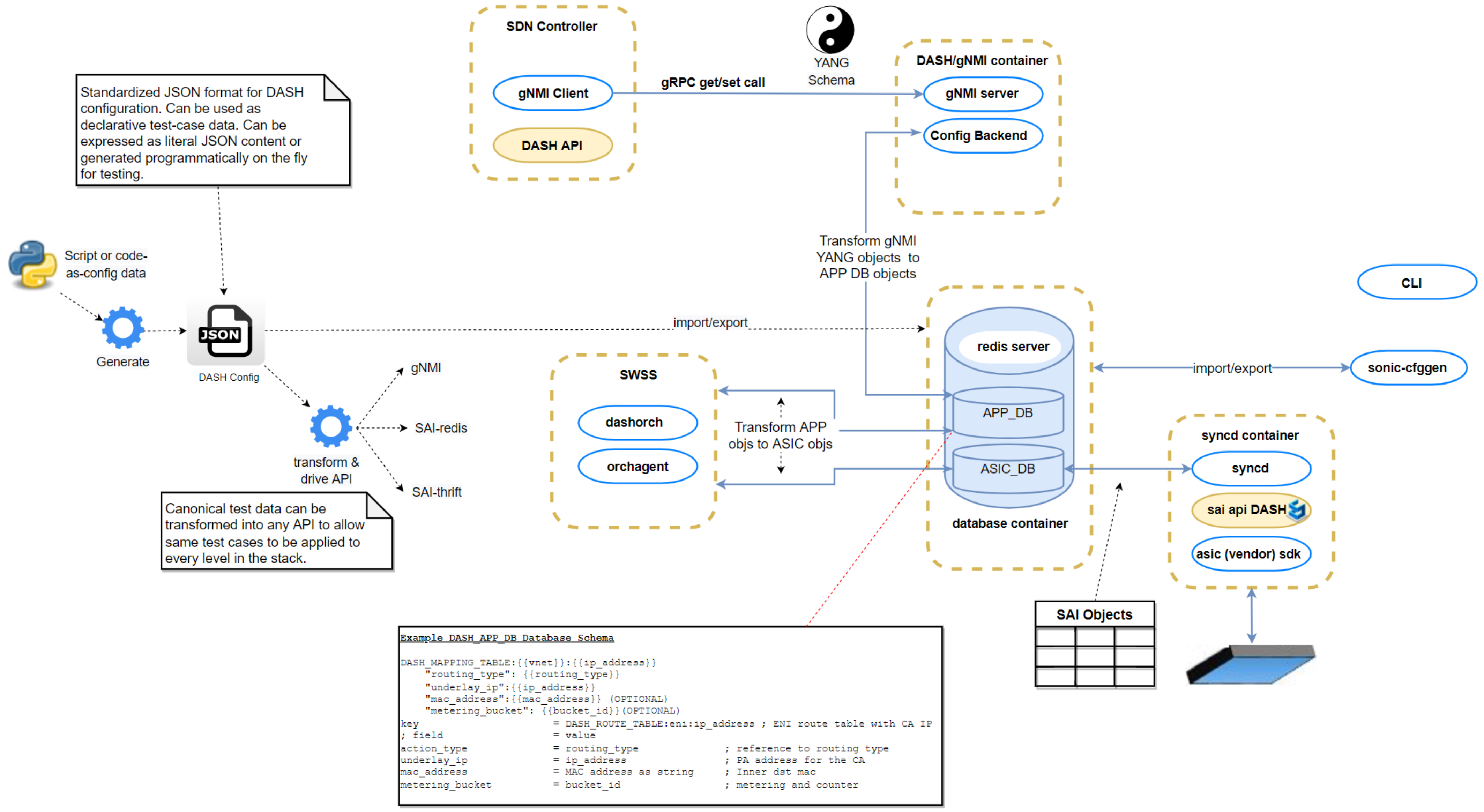
Q&A, Feedback?

Backup Slides

Test Methodology Evolution

What	**Existing tools**	**…Adding New Tools**
Test framework	Unittest	Pytest
DUT APIs	saithrift	saithrift, sairedis, gNMI (future)
Dataplane	PTF (Scapy based)	PTF and/or **snappi** (open traffic generator)
Coding style	Concrete use of sai_thrift APIs	Abstract, config data-driven; underlying APIs taken care of by framework
Granularity	Direct access to each API and data type, allows arbitrary API usage	Config + helpers hide the API details (but also discourages direct access)
Expertise	Requires intimate knowledge of APIs and data types	Config data easy to understand, API knowledge not required
Packet testing: Speed & Scale	“Packet-at-a-time” testing, speed is limited	Packet-at-a-time or flow based, speed up to full line rate
Config scaling	Ad-hoc coding, limited by ingenuity & Scapy limitations	Built-in handling of large static configs or on-the-fly config generator

Recap – Schema Relationships



Standardized JSON format for DASH configuration. Can be used as declarative test-case data. Can be expressed as literal JSON content or generated programmatically on the fly for testing.

Canonical test data can be transformed into any API to allow same test cases to be applied to every level in the stack.

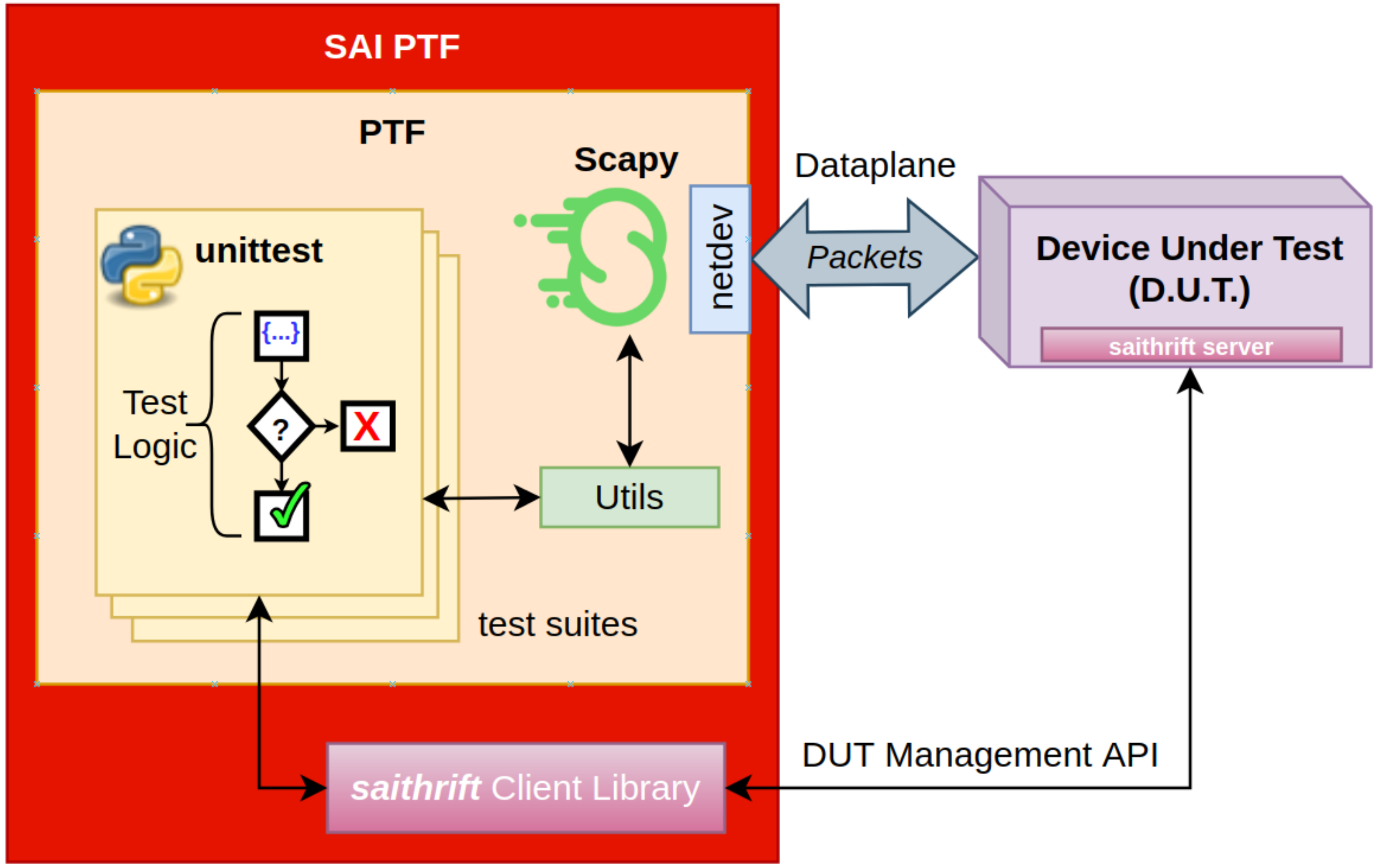
```
Example DASH_APP_DB Database Schema

DASH_MAPPING_TABLE:{{vnet}}:{{ip_address}}
    "routing_type": {{routing_type}}
    "underlay_ip":{{ip_address}}
    "mac_address":{{mac_address}} (OPTIONAL)
    "metering_bucket": {{bucket_id}}(OPTIONAL)
key                         = DASH_ROUTE_TABLE:eni:ip_address ; ENI route table with CA IP
; field                     = value
action_type                 = routing_type                 ; reference to routing type
underlay_ip                 = ip_address                   ; PA address for the CA
mac_address                 = MAC address as string        ; Inner dst mac
metering_bucket             = bucket_id                    ; metering and counter
```

KEYSIGHT

https://github.com/Azure/DASH/blob/main/documentation/general/design/dash-sonic-hld.md#figure---schema-relationships

Background: PTF/SAI-PTF

- **Unittest** is a Python framework for generic software unit tests. Developers write “test suites” with pass/fail outcomes.
- PTF was created to test dataplanes. It combines the unittest framework with Scapy, a popular software traffic generator/capture tool, plus various utilities to make dataplane tests easy to write. It does not include a DUT configuration API or transport.
- **SAI-PTF** is PTF with an Apache Thrift RPC transport layer plus Python client libraries for SAI configuration.



KEYSIGHT

KEYSIGHT

Thank you