WHITEPAPER

Apstra AOS Architecture Overview

Sasha Ratkovic, CTO, Founder

apstra®

WHITEPAPER

Introduction: Apstra AOS Is Designed to Solve the Hardest Data Center Problems

Building, deploying, operating, managing and troubleshooting a data center network can be difficult, expensive, and resource-intensive. According to a 2019 Gartner report, simplifying IT infrastructure is the number 1 strategic priority for businesses today. Apstra's operating system (AOS) has been purpose-built to automate the full life cycle of the data center network.

In this paper, we discuss how the architecture of the Apstra operating system works to solve some of the most difficult problems faced by data centers today: composition, reliable change, extensibility and scalability.

AOS Architecture Goals: Composition, Reliable Change, Extensibility, Scalability

There are four main challenges facing data centers today:

- Composition
- Reliable Change
- Extensibility
- Scalability

Composition

Putting together a coherent, smooth working system with pieces of infrastructure from different vendors, and with different capabilities, can be incredibly difficult. Apstra's operating system (AOS) creates a coherent whole - called a **reference design** - that enables an operator to deliver reliable and easy-to-consume services.

Reliable Change

Change is a constant in the data center. Change can come from an operator trying to add new services or expand capacity, or from the infrastructure in the form of failure conditions. Either way, operators have to find better ways to deal with change. According to a variety of studies in recent years.[1] 65-70% of network outages are caused by human configuration error while executing a planned change.

Extensibility

In order to encourage innovation and accommodate the inevitable rapid change in technology, data centers need to be able to add new capabilities and functionality easily and smoothly. As technology evolves, vendors innovate and come up with new features and capabilities that operators need to leverage in order to stay competitive. This evolution causes a design-time change resulting in a need for new reference design. In other words, your composition has to be extensible to allow for further innovation or to address new requirements.

Scalability

You want to be able to address all of this at scale. Data center networks need to be able to grow and accommodate innovation and increasing complexity.

The primary goal of the Apstra operating system architecture is to address these issues directly, as we describe in the following pages.

[1] Examples include: https://www.veriflow.net/survey/; https://www.computerweekly.com/news/2240179651/Human-error-most-likely-cause-of-datacentre-downtime-finds-study; https://www.networkworld.com/article/3142838/top-reasons-for-network-downtime.html; https://www.ponemon.org/library/national-survey-on-data-center-outages.

The Main Challenge: Composition

The main challenge facing operators running compute/network/storage infrastructure today is composition. Composition is the ability to create a coherent whole that is bigger than just the sum of its functioning components and to sustain that coherency over time in the presence of inevitable changes — all the while delivering network services to consumers. At its core, the Apstra operating system is designed to rise to the composition challenge.

Modern data centers are scale-out computers. They need an operating system that provides functionality analogous to what a host operating system provides on a single machine today: resource management and process isolation.

Compute virtualization already does this for a single compute. But for data-center-as-a-scale-out compute, a data center operator first needs to compose it — only then can you provide resource partitioning.

> At its core, AOS is designed to rise to the composition challenge.

Why Is Composition So Difficult?

There are two main reasons composition is such a challenge. The first is that the elements that you are using to compose a coherent whole may come from different vendors, have different capabilities, and may only be available via different APIs.

Second, you may need to compose your infrastructure to deliver a variety of services, each with multiple functional aspects such as reachability, security, quality of experience, and availability.

As you instantiate multiple service instances, interactions between these components can cause service outages — unless you have a firm understanding of the mapping of these services into enforcement mechanisms. And as the capabilities of your infrastructure evolve over time, you need to be able to leverage and introduce new innovation without breaking the existing systems.

AOS Addresses the Challenge with Reference Design

The key concept that Apstra architecture uses to address the composition challenge is reference design. Reference design is a behavioral contract that defines how the business intent of the user is mapped to enforcement mechanisms and what expectations must be satisfied in order to deem the intent fulfilled.

Reference design governs:

- The roles and responsibilities of physical and logical components
- How services are mapped to enforcement mechanisms
- The expectations that need to be met (i.e. situations to watch)

In a reference design, the roles and responsibilities of the physical and logical elements are well defined, which in turn binds the scope of modeling and enables the specification of a minimal, yet complete, model.

Reference design also governs how the intent is mapped into enforcement mechanisms. Understanding of this mapping enables automation of troubleshooting and provides powerful analytics rooted in knowledge of how the system is composed, as opposed to reverse engineering what is happening in your network.

Figure 1: Reference Design Is Behavioral Contract Which Governs

What are the roles and responsibilities of physical and logical components?

Reference design binds the scope and enables specification of minimal yet complete models

How are services mapped into enforcement mechanisms?

You can have different reference designs for different domains (data center, campus, edge)

What are the expectations (situations to watch) that need to be met?

You can have standard or snowflake reference designs

In different reference designs, the same intent may be enforced with different, possibly newer, and more innovative mechanisms. Understanding this mapping enables root-cause identification, which points operators directly to the cause of service-affecting problems.

Reference design also specifies the expectations to be validated. For example, reference design may stipulate that each leaf should have a border gateway protocol (BGP) session with each spine. This way, missing and unexpected BGP sessions are easily identifiable.

Defining Reference Design Requires Expertise

Defining a reference design is a value-added activity that must be completed by a networking expert. AOS makes expertise artifacts an explicit part of the system and not something that lives only in the head of the expert. This allows expertise to be modeled and inserted into the system, as opposed to being hardcoded.

Knowledge is Power: Dealing with Change Reliably

Change is a constant for data centers, so dealing with change reliably is always top of mind.

Change comes from two places:

1. **Operator's Intent.** An operator may want to add a new service such as virtual network, add/remove a resource from the infrastructure, or make a change in some policies. The operator has control over a change as it's initiated. Apstra's operating system uses stateful orchestration to enable an operator's intent reliably.

It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to *change*.

-Charles Darwin, 1809

2. **Failures in Managed Infrastructure.** Change can also come from managed infrastructure in the form of failures (e.g., excessive packet drops, traffic imbalances). An operator doesn't have control over this change, and the volume of operational conditions is often overwhelming. AOS provides intent-based analytics to help operators deal with changes in the operational state.

Keys to Reliable Change: Stateful Orchestration and Intent-Based Analytics

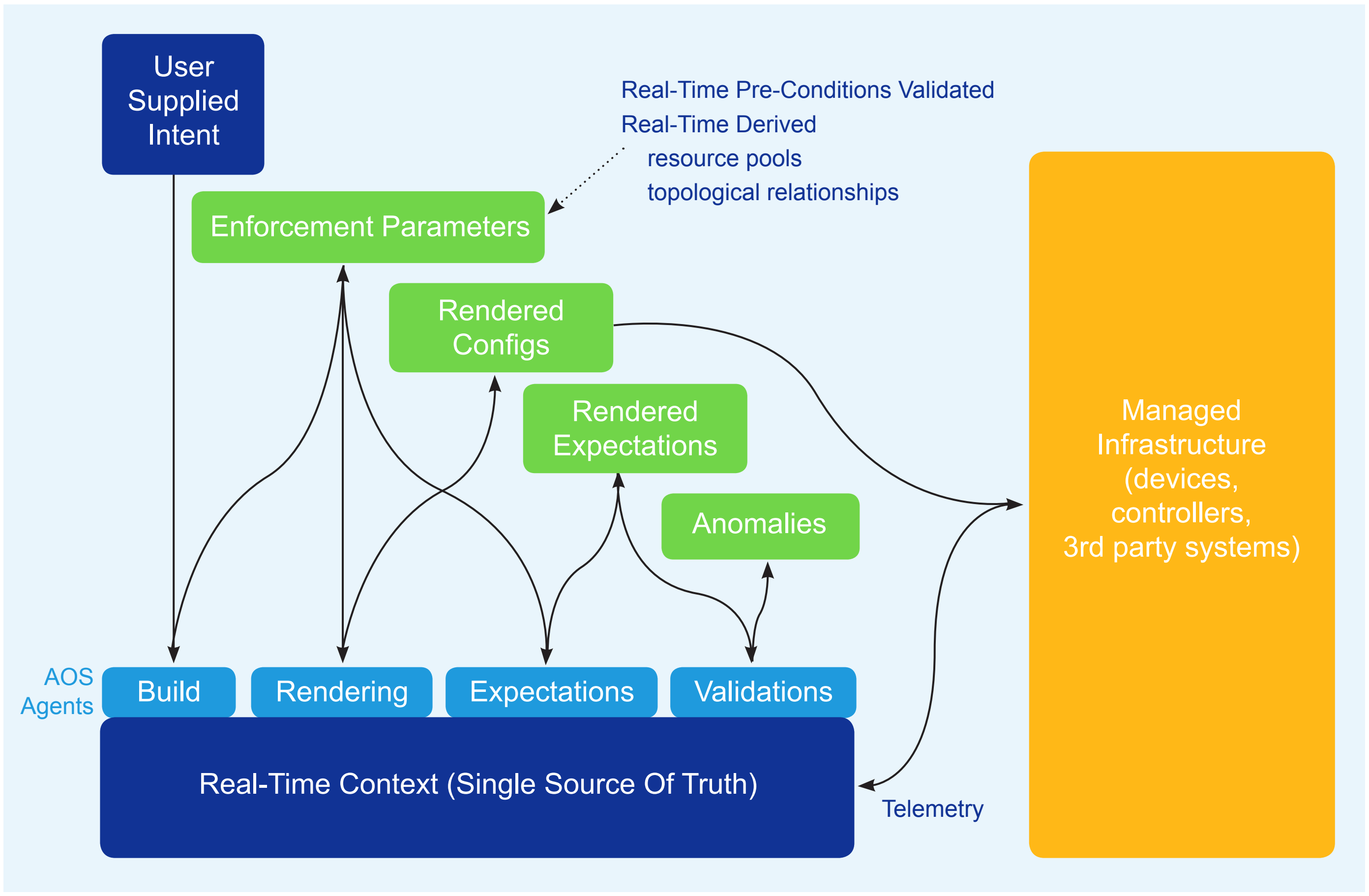
Dealing with both types of change reliably depends on your knowledge of the state of your infrastructure. Two conditions must be met:

1. Knowledge has to be interpreted in context – meaning that the relationship between conditions and expectations is understood –as described in the Context Model.
2. Knowledge has to be timely, meaning that it reflects current conditions, as described in Real-Time Aspect.

Stateful Orchestration

Stateful orchestration makes changes coming from the operator reliable using intent. A stateful orchestration flow of actions can be seen in the following:

Figure 2: Stateful Orchestration Flow



You want to make a change on top of your existing, fully functional system. With stateful orchestration, the user supplies only the intent for change. This is done in an implementation-agnostic way, which makes intent-specification simpler and less prone to errors. A complete set of steps executed during stateful orchestration is as follows:

Step 1: Real-time pre-condition validation. Is this new request going to violate some policies? For example, are you allowed to create this virtual network, or is it going to create some security holes? Or can you put a device into maintenance mode knowing that some other devices are already offline and taking this device offline is going to make your system vulnerable? With stateful orchestration, all these questions are asked and validated in real-time against the context graph.

Step 2: Real-time enforcement parameters derivation. In order to implement intent, specific parameters for appropriate enforcement mechanisms will have to be supplied and activated on specific managed elements. Reference design, which spells out how intent is to be implemented, along with understanding the current context, enables AOS to automatically perform real-time enforcement parameters derivation. There is no possibility of an operator accidentally inserting a wrong command, interface, or IP address or VLAN.

Step 3: Real-time multi-config deployment. AOS performs real-time multi-config deployment, which automates deployment of configuration on possibly multiple elements which may have different vendor- and technology-specific APIs.

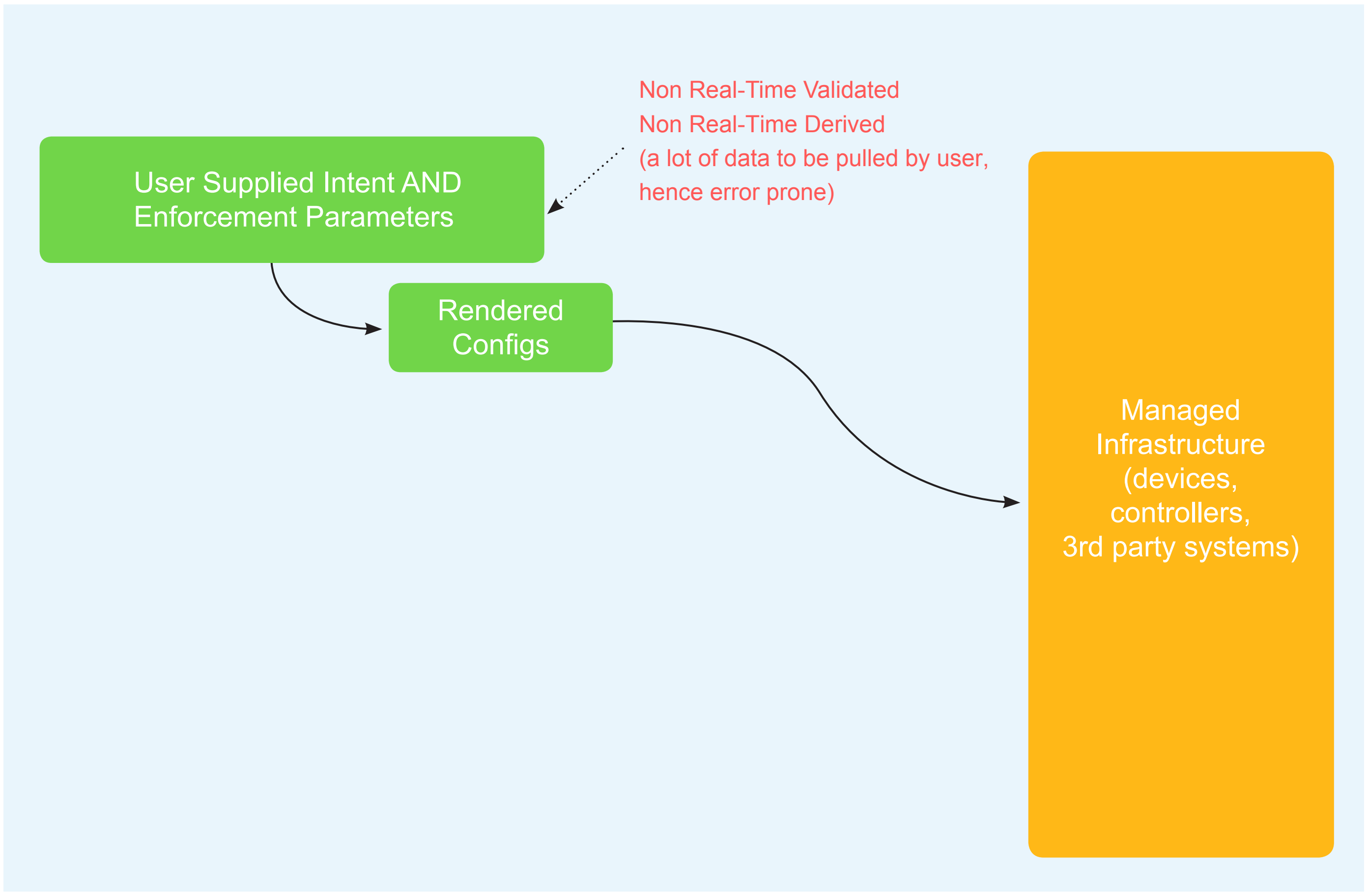
Step 4: Real-time expectations generation. Reference design acting as behavioral contract allows AOS to perform real-time expectations generation, which describes conditions that need to be met in order to declare if the outcome has satisfied the intent.

Step 5: Real-time expectations validation. AOS then triggers a collection of telemetry and context-enabled operational analytics to perform real-time expectations validation.

Step 6: Validated Service Outcome. At the end of this process, the user can observe a validated service outcome in unambiguous terms. Every change in intent or expected operational status is reflected in the context model in real-time, and any component that needs to be aware of the change is also notified about it in real-time. AOS extracts relevant knowledge from raw operational data using Intent Based Analytics.

With stateless orchestration, many of the steps are missing (as shown in figure below). Config deployment outcome is what stateless orchestration is typically all about — pushing configuration to possibly multiple systems and validating that the configs have been accepted by managed elements. And this is where stateless orchestration typically stops; there is no notion of service expectations or validation that these expectations are met. Automated service-outcome validation is absent and, instead, separate systems present a user with a “single pane of glass” regarding the raw operational state. It is up to the user to perform visual discovery to find out if the outcome has been achieved. This single pane of glass usually includes too much information and lacks implementation-specific context, which makes visual discovery extremely difficult and subjective.

Figure 3: Typical Stateless Orchestration


Non Real-Time Validated
Non Real-Time Derived
(a lot of data to be pulled by user, hence error prone)
User Supplied Intent AND Enforcement Parameters
Rendered Configs
Managed Infrastructure (devices, controllers, 3rd party systems)

Intent-Based Analytics

Intent-Based Analytics (IBA) helps operators deal with operational status changes in their infrastructure by extracting knowledge out of raw telemetry data. As mentioned before, you must have real-time query-able context before you can use IBA.

Knowledge extraction consists of two steps:

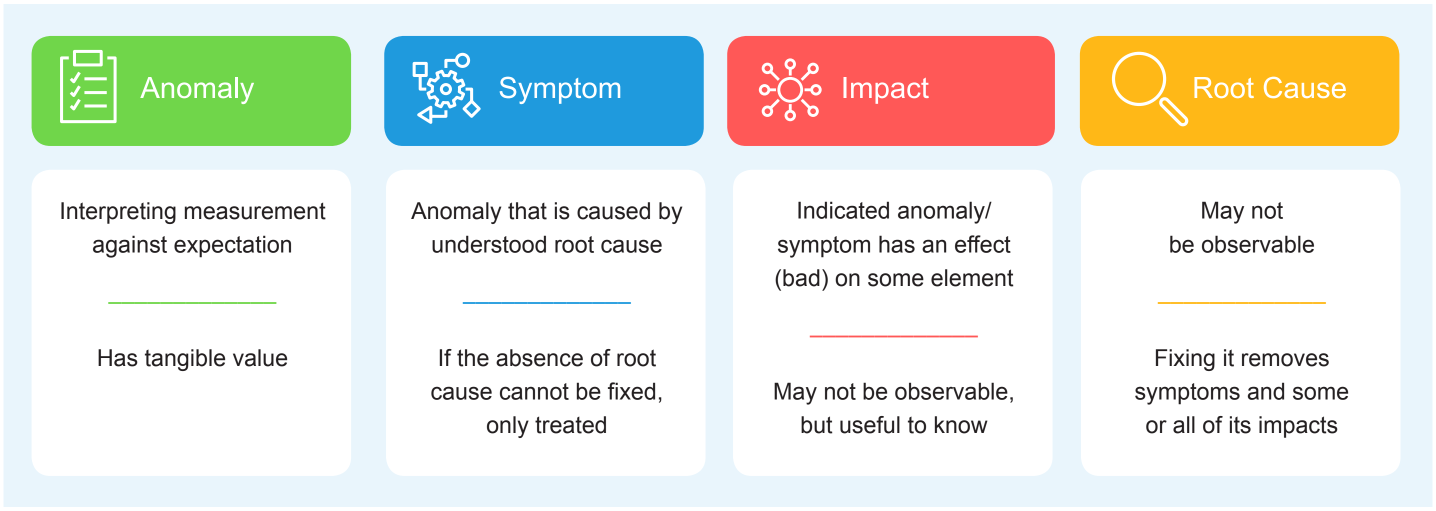
1. **Detection of conditions of interest** (situations to watch). Conditions detection is done by **IBA probes**.

2. **Automation of classification of conditions of interest and deriving relationships between them.** Conditions vary in their semantic content, in other words, some of them are more important than others. It's important to understand the relationships between conditions so that you can pinpoint important actionable conditions (root causes) and understand which conditions are merely consequences that will disappear when important root causes are taken care of. Condition classification and causality derivation is done by the Root-cause identification (RCI) component of IBA.

Modeling Conditions of Interest

IBA models conditions of interest using four categories: anomaly, symptom, impact and root cause.

Figure 4: Categorization of Conditions



Anomaly	Symptom	Impact	Root Cause
Interpreting measurement against expectation	Anomaly that is caused by understood root cause	Indicated anomaly/ symptom has an effect (bad) on some element	May not be observable
Has tangible value	If the absence of root cause cannot be fixed, only treated	May not be observable, but useful to know	Fixing it removes symptoms and some or all of its impacts

Anomalies

Anomalies essentially represent an interpretation of measurement, or some aggregate of it, against some expectation, and as such have more tangible value than simple measurement.

Symptoms

Symptoms are anomalies caused by a well-understood root cause. They are typically easily observable, but they cannot be fixed and can only be treated. Like giving aspirin to a patient with a high fever, it treats the symptom. Unless you treat the illness that is the root cause for fever, the fever will return. Importantly, a symptom will disappear when the root cause is fixed, so symptoms are not actionable; they are simply useful for diagnosing the root cause fault.

Impacts

Impacts indicate that something has happened as a result of an anomaly. Impacts may not always be observable, but they are useful to know about. For example, you may want to know that an important customer is going to be impacted by a failure of a device or excessive packet loss before they call you to complain.

When impacts are not observable, they can be calculated based on the knowledge of the intent and the enforcement mechanisms used to implement the intent. Understanding impacts helps operators prioritize which root causes and anomalies have the most impact on the operation.

Root Causes

The most important of all conditions are root causes. They may not be always observable which makes them difficult to diagnose, but they cause many symptoms and impacts. They are actionable as fixing the root cause results in the disappearance of associated symptoms and impacts.

IBA Probes: Knowing What You Know

A famous quote applies to data center networks: “It ain’t what you don’t know that gets you into trouble. It’s what you know for sure that just ain’t so.” Intent characterizes and formalizes the normal — what we do know. Intent-Based Analytics ensures that what we know is, in fact, as it should be.

IBA probes are responsible for detecting conditions of interest and are driven by a behavioral contract that is part of the reference design. IBA probes fetch data, apply some processing, and then compare the result against expectations. An IBA probe is essentially a configurable data-processing pipeline that allows users to set up conditions of interest (i.e., situations to watch).

“

It ain’t what you don’t know that gets you into trouble. It’s what you know for sure that just ain’t so.

Data Sources and Queries

The initial stages of a probe are typically data sources. They are responsible for fetching the raw telemetry data. The data source stage’s configuration includes a **query** that can express precisely which data to collect — a very powerful feature.

For example, say an operator is interested in analyzing an equal-cost multi-path routing (ECMP) imbalance that applies only to fabric interfaces and there were reports that a specific operating system version introduced a bug in the ECMP hashing algorithm. A query can express the need to collect interface counters on every top of the rack switch, but only for fabric interfaces and for switches that are running particular version x.y.z of switch operating system. Now that this situation-to-watch is set up, the operator does not need to worry about the changes. If a new fabric link is added to a switch that matches the criteria, it will be automatically included in the analysis. If a new switch is added, it will be automatically included. If someone upgrades another switch to version x.y.z, it will be automatically included. The IBA probe requires no maintenance.

Stage Processors

In many situations, the operator is not interested in instantaneous values of raw telemetry data, but rather in an aggregation or trends. IBA contains stage processors aggregate information such as calculating average, min/max, standard deviation, and so on. An operator can then compare these aggregates against expectations so that they can identify if the aggregate metric is inside or outside of a specified range, in which case, an anomaly is flagged.

The operator may then want to check whether this anomaly is sustained for a period of time exceeding a specific threshold and flag the anomaly only when the threshold is exceeded in order to avoid flagging anomalies for transient or temporary conditions. The operator can achieve this by simply configuring a subsequent stage to contain what’s called a time-in-state processor.

Beyond Numerical Data

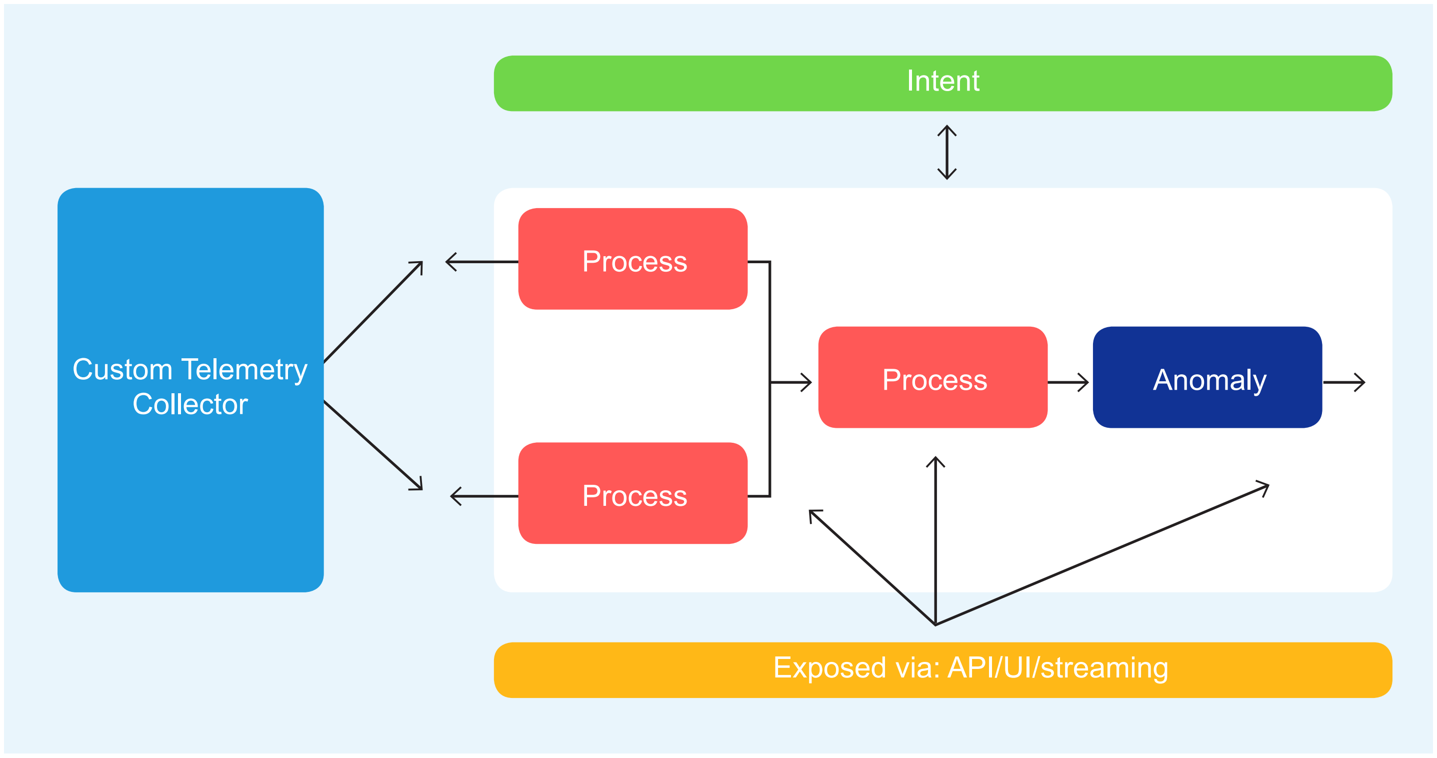
Probes do not operate only on numerical data. For example, they can be used to validate the correctness of control and data plane. An operator can, for example, configure a data-source processor with a query to construct an expected routing or forwarding table, given the intent.

In an EVPN environment, the intent will contain information such as the virtual networks that exist, where their endpoints are, and information about enforcement mechanisms. This expected table can then be compared against the one coming from the telemetry and anomalies flagged when a mismatch is found.

A probe can also be configured to track sudden changes in the sizes of forwarding and routing tables and alert when trends are not as expected. The “what is expected” threshold can also be calculated dynamically from the intent as it can be a function of the number of virtual networks, the number of endpoints, the number of VTEPs, the number of VTEPs with some issues, and so on. The possibilities are endless.

Probes can be declaratively created, activated, and deactivated with a simple REST call or with GUI.Probe activation also serves as a trigger for telemetry. In other words, specific telemetry is collected only if there is a probe interested in it, so probes essentially serve as telemetry collection configuration mechanism. Queries in data-source processors make configuration as granular and precise as needed, eliminating "data hoarding disorder," which happens when tons of data is collected without a clear idea about what to do with it. This in turn drives the costs of storing and processing data through the roof.

Figure 5: Telemetry Collection



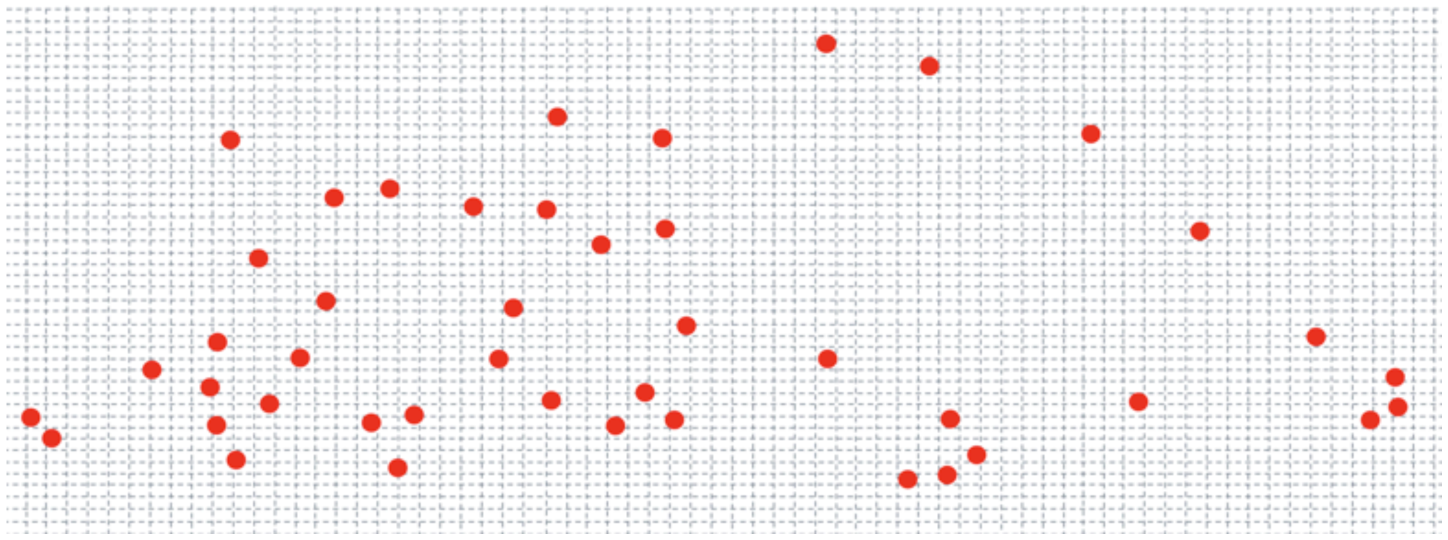
Once a probe is configured, the output of every stage is available via API, UI and streaming endpoints. AOS comes with a set of built-in probes, and there is a repository of open source probes on GitHub. And new probes can be created from scratch. For more information, review Understanding Intent-Based Analytics: A Technical Deep Dive.

Root-Cause Identification

Root-Cause Identification (RCI) is a mechanism to automate the classification and derivation of causality relationships between conditions that are identified in the infrastructure. The main benefit of RCI is surfacing root cause conditions that require operator action from the sea of non-actionable conditions that are merely consequences of root cause faults.

For example, suppose an operator has instrumented the infrastructure well and is observing a log of anomalous conditions on a "single pane of glass" console (see Figure 6). The red dots indicate different types of conditions scattered across the infrastructure occurring in many different elements.

Figure 6: Single Pane of Glass Console



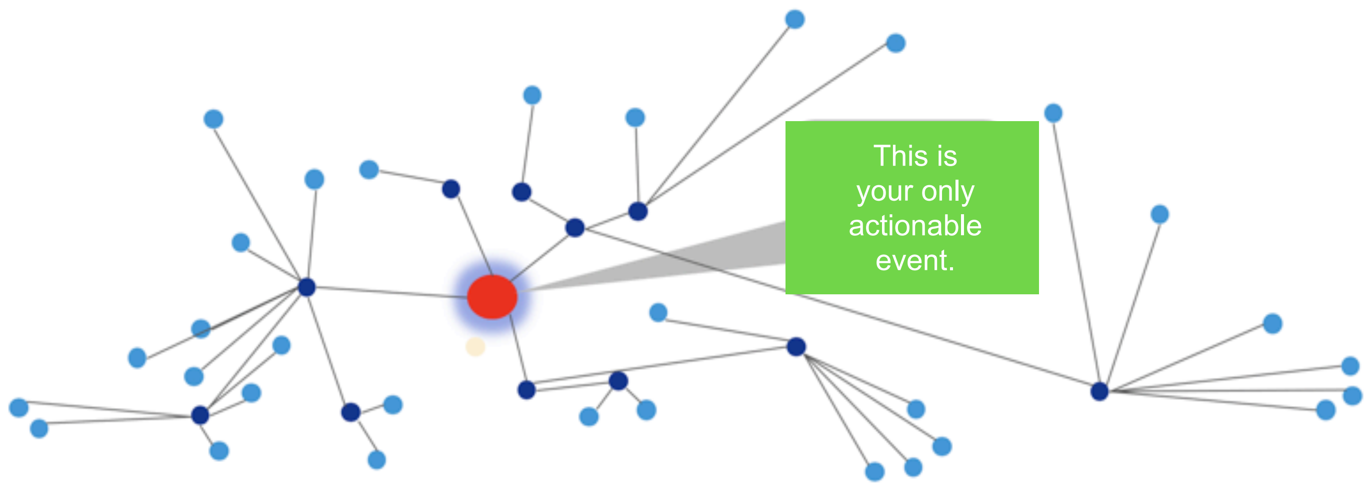
The problem with this picture is that it is not actionable — there is a lot of noise. In the absence of classification, each of the conditions are vying for the operator’s attention, but which one needs to be acted on?

This is where RCI comes into play. Armed with the knowledge of context, which says what the service is, how it is implemented and mapped into enforcement mechanisms, and how the elements in the managed infrastructure are related, RCI identifies and classifies the root cause(s) and related symptoms and impacts.

The results of the analysis using RCI are (see Figure 7):

- The root cause was a memory leak on switch called “spine_1” (Root cause: large red dot)
- Which caused the Out-Of-Memory process killer to act and kill a few processes, one of which was a routing process (Symptoms: dark blue dots)
- Which in turn caused BGP sessions to be down on this and the peering devices along with missing expected routing table entries (Symptoms: dark blue dots)
- Which caused endpoints belonging to customers X and Y to experience connectivity issues (Impacts: light blue dots)

Figure 7: Root-Cause Identification Console



RCI automates the complex mental process involved in a “single pane of glass,” which overloads the operator with mountains of information and leaves it up to them to perform visual discovery and correlation. Instead, RCI presents an operator with a simple pane of glass, which simply identifies the actions needed.

Context Model

Having all the data doesn’t mean you have all the answers. And collecting huge amounts of data without automating the extraction of knowledge is an invitation to an OPEX explosion, as you will have your experts spending their valuable time working on making sense out of data. Data in itself has limited value unless it gives you knowledge or it gives you answers to the right questions.

Raw telemetry data is typically stored in key-value stores, which are well-suited for horizontal scaling and sharding. But they don’t support queries other than simple key lookups. So, to extract knowledge you would need to build support for queries AND have a context to construct these queries.

On the other hand, SQL data stores do support complex queries. The problem is that SQL tables are designed around anticipated queries, which are created during design. Should you want to ask some other question at run time, you may de-normalize the database or the queries may end up with a lot of joins, which will cause performance to grind to a halt. But extraction of operational knowledge is all about queries that come up at run time. This is where graph-based implementation of contextual data shines — supporting arbitrary complex queries with a large number of “joins,” at run time. The graph model can also be easily extended, as you simply add more instances of basic building blocks, nodes and relationships.

Let’s take a closer look into what goes into this context as it will help explain the power unleashed by the queries.

- **Design Artifacts:** First, the intent-graph context contains design artifacts. For example, if you are designing a data center network, it may contain the desired oversubscription factor, desired number of servers, and desired high-availability configuration (single-attached, dual-attached).
- **Resource Allocation:** It contains decisions related to allocation of resources. This governs whether you are managing your infrastructure as “cattle” or “pets” http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

- **Isolation Policies:** It also specifies isolation policies — Is it allowed to reuse certain resources (IPs, ASNs, VLANs)?
- **Segmentation Policies:** It contains segmentation policies, specifying which endpoints and workloads can talk to each other and under what conditions.
- **Enforcement Information:** It also contains enforcement information about where (on which physical or virtual appliances) and how (using which mechanisms) the segmentation and reachability policies were actually implemented.
- **External Reachability Policies:** In case of the data center, it may also contain external reachability policies, specifying which inside endpoints can be seen by the outside world, and vice versa. It also contains which tenants own which segments and what were the promised service levels to each of them, what were the service level objectives.
- **Business Perspective:** It also covers the business perspective, such as the service-level agreements and how much it will cost you if you don't meet them.

8-D View

Even though all these artifacts are represented in a single graph, that single source of truth is logically a single multi-dimensional space, with the following at least eight dimensions (derived from this example, but possibly even more):

- Design
- Resources
- Isolation
- Segmentation
- Enforcement
- External reachability
- Service levels
- Business perspective

As a result, when things go wrong, you can have a single query operate across this 8-D view and answer complex questions such as: "Given the recent failure condition on an element, are my design goals still valid? Are the correct resources used on correct enforcement points, with desired isolation policies? Does it impact segmentation policy in any way? Does it honor the service-level objective? And what is the price I am paying for this failure?"

Existence of this 8D view is not a "nice to have" — it is a prerequisite for reliable operation. With AOS the 8D view is built in. Without AOS, you have to reconstruct this 8D view with a very complex layer, spanning multiple sources of truth and tribal knowledge that exists in the minds of your experts. Building this layer, in an environment where individual sources of truth were not built with integration in mind and have different semantics and behaviors is undifferentiated heavy lifting in the best case, and an unmanageable nightmare in the worst case.

Real-Time Monitoring and Notification

AOS is all about extracting knowledge about the intent and the resulting operational state of your infrastructure. This knowledge is power if it is timely, i.e., if it reflects current conditions. At the heart of AOS is the ability to configure live queries, which enable clients to subscribe to conditions of interest and get notified real-time when the conditions are met. And conditions in this context are related to both intent and operational state. This is an absolute prerequisite for dealing with changes reliably.

As you will see in the Architecture Overview, the same pub/sub paradigm that is presented at the user level is supported by an equivalent low-level pub/sub mechanism that is implemented by the distributed data store acting as a data-centric logical communication channel for the AOS processes that implement application logic.

The hurdles to a self-operating network are more organizational or people-acceptance than technological, as the required functionality already exists.

Self-Operation

Last, but not least, there is a need to automate the reaction to some events in order to remediate problems, log them for forensic analysis or do a next-level drill down to collect telemetry to help with root-cause analysis.

An automated reaction:

- Reduces risk, as remediation parameters are automatically derived from the up-to-date single source of truth and are not subject to misconfiguration or stale data
- Improves customer experience, as the remediation happens in a timely manner
- Reduces operational cost, as it eliminates the need for manual troubleshooting and manual execution of the remediation playbook

Ultimately, an automated reaction enables network self-operation and self-healing. The hurdles to a self-operating network are more organizational or people-acceptance than technological, as the required functionality already exists.

Extensibility: Future-Proofing the Data Center Network

Extensibility has three dimensions:

1. **Extending reference design**, which governs how the pieces of infrastructure are composed together to deliver on the intent. This includes plug-ins that allow modifications of the graph model as well as modifications related to how the intent is mapped into enforcing mechanisms in the infrastructure
2. **Analytics extensibility**, which is related to the definition of new conditions or situations to watch. This allows a user to define new validations as well as how to classify and relate them
3. **Flexible service APIs**, which provide the ability to extend the top-level service definitions

Reference Design

As mentioned earlier, reference design is a behavioral contract that defines how intent is mapped to enforcement mechanisms and what the expectations to be satisfied are in order to deem intent fulfilled.

Any aspect of this contract can be modified or extended. New node and relationship types can be defined. Configuration templates can be altered. Resource allocation mechanisms can be altered. In depth explanation of reference design extensibility is covered in a separate document.

Analytics Extensibility

New analytics functionality can be introduced via two mechanisms:

1. **New IBA probes** can be defined that detect new conditions of interest. They may include new telemetry collectors as well as condition-specific data processing pipelines. Probes can also be published and subsequently imported from a public repository.

2. **New RCI models** can be defined and loaded into a system. An RCI model is essentially a mapping of a new type of a root cause to a set of symptoms that it creates. Once this model is defined and loaded, AOS automates root-cause identification based on observed symptoms.

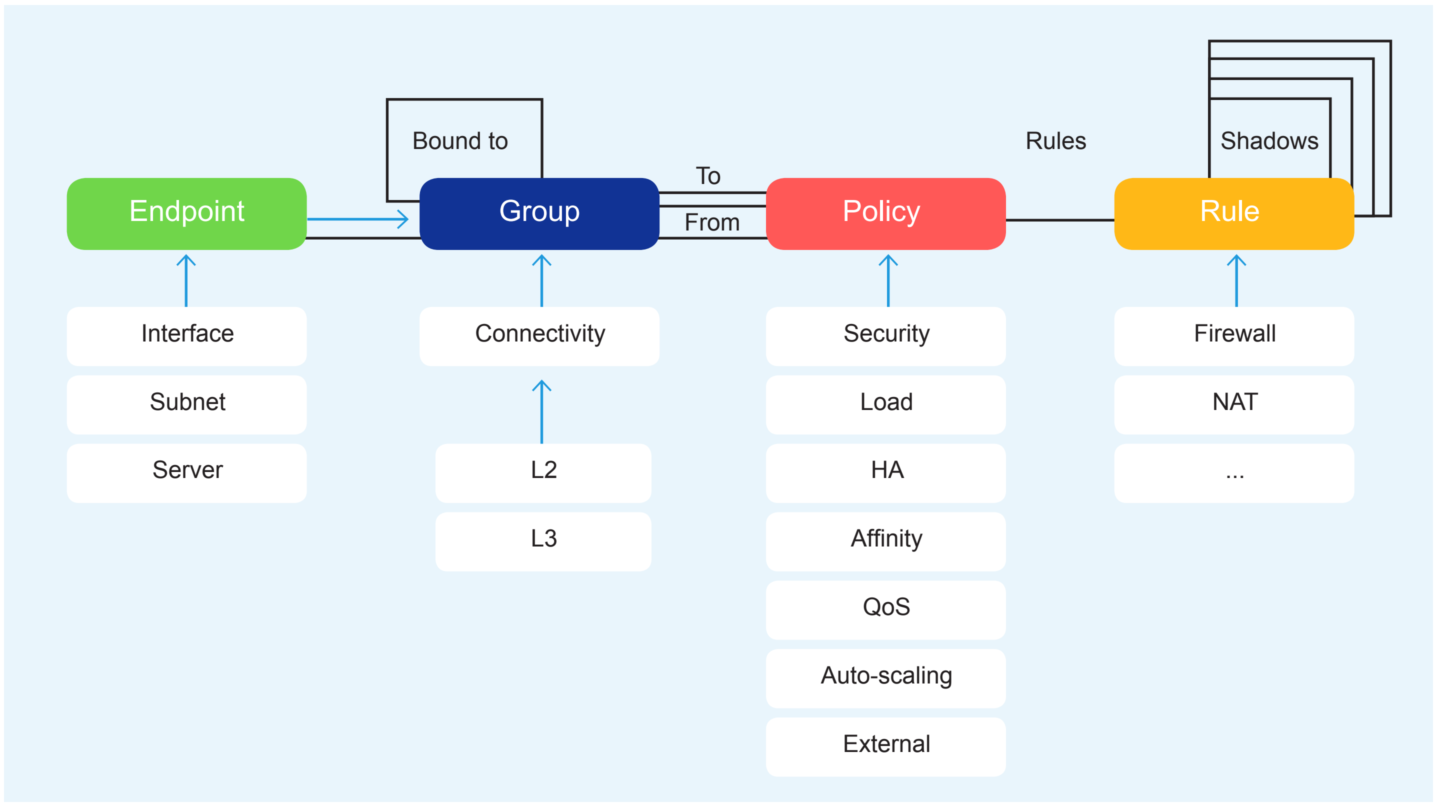
Service APIs

AOS exposes service-level APIs in the form of group-based policies, which allows flexibility in supporting a wide range of services and policies in an implementation-agnostic way.

With group-based policies, intent is expressed as a graph, representing endpoints that are placed into groups (member relationships) with the purpose of expressing intent for some common behavior. Policies are instantiated and related to groups or individual endpoints to define that behavior.

Policies can relate to a group in a directional manner (from/to relationship) or non-directional (applies-to). Policies are a collection of rules. Rules may have a next-rule relationship when ordering between the rules is important. Groups can be composed of other groups (hierarchy). Groups can also have a relationship to another group to express some constraints (such as “these endpoints/ groups are behind these groups of ports”). Endpoints, groups, policies and rules can be thought of as building blocks for expressing connectivity intent.

Figure 8: Group-Based Policies



Endpoints are elements of your infrastructure that are subject to policies and as such are quite general in their construction. They can represent an interface (physical or virtual), server, VM, container, or application endpoint. The blue arrows in Figure 9 indicate "logical" inheritance. Endpoints contain parameters that define them more precisely (e.g., interface name, port number, server s/n and hostname, VM uuid, container IP, application protocol, UDP/TCP port number). Endpoints can also represent elements not managed by AOS (external endpoint) and as such are used to express constraints from the external systems with which AOS needs to interact (e.g., external router IP/ASN).

Endpoints are placed into groups. Any given endpoint can be a member of multiple groups, each expressing different aspects of intended behavior. For example, a group could indicate:

- A set of servers in the "East" region
- A group of servers that are to be used for load balancing
- A group of servers that form a redundancy group

Groups can also have overloaded semantics. For example, endpoints that are members of the L2 domain group are members of an L2 broadcast domain (subnet). Similarly, endpoints that are members of the L3 domain have L3 reachability.

Groups can also be composed of other groups. Endpoint members of the group may be explicitly instantiated or there may be a dynamic membership specification as part of a group definition where endpoints are implied from the specification present in the group.

For example, an L3 domain may have CIDR or subnet as a property and therefore all the endpoints with IPs in that range/subnet are implicitly members of the group. In that case, the L3-to-the-server reference design specifies a number of external/internal subnets, and even though endpoints (containers) are not explicitly specified and managed in AOS, it is implied that containers do belong to the specified L3 domains. But in order to model granular segmentation, we are also going to introduce explicit specification of container endpoints and the servers they are hosted on.

A policy defines common behavior and can contain parameters to define that behavior precisely. Policies can be applied to the group in a non-directional manner) or direction can be indicated when needed, such as in security policies. Examples of specific policies include security, load balancing, HA, affinity (e.g., the desire to collocate endpoints or to distribute them), QoS, etc.

Policies can contain rules when needed. Rules typically follow the "condition followed by an action" pattern. For example, in the security policy, "match" is a condition statement, and action is "allow/deny/log."

Scalability: Growth Without Pain

AOS supports network-transparent distributed state access and management, while parallel execution is supported by the separate processes. Real-time execution is supported by an event-driven asynchronous model of execution together with real-time scheduling of execution. Efficiency and predictability are supported by compilation through C++ as an intermediate language to achieve machine-level efficiency.

There are three dimensions to scaling:

Scaling the State

The first dimension is scaling the state. The AOS data store scales horizontally, by adding more high-availability (HA) pairs of servers. In AOS, intent and telemetry data stores are separated and can scale independently as needed.

There is also a provision for hierarchical data stores, where a top-level data store subscribes to (synchronizes with) only the required (as specified by designer) subset of state from the next-level data stores, as required for correlating the state across data stores.

Scaling Processing

The second dimension of scaling is processing. AOS can launch multiple copies of processing agents (per agent type) if and when required that will share the processing load. More agents can be added by adding more servers to host them, and an agent's lifecycle is managed by AOS.

AOS state-based pub/sub architecture allows agents to react (provide application logic) to a well-defined subset of state. Coverage of the whole intent is done through separate agents delegated to dealing with different subsets of state. This means that when there is a change in the intent or operational state, the agent's reaction is to "incremental change" and is independent of the size of the whole state.

AOS employs the traditional approach to deal with scale and associated complexity — decomposition. The "everyone knows everything" approach doesn't scale. So, you have to distribute the knowledge about the desired state and let each agent determine how to reach that state, so you avoid needing centralized decision making. AOS support for live graph queries implies that clients such as UI can ask for exactly what they want and get exactly what they need and nothing more, allowing granular control of the amount of data to be fetched from the back end.

Scaling Network Traffic

The third dimension is scaling network traffic. Communication between the agents and data store is using an optimized binary channel, thus significantly reducing the amount of traffic compared to text-based protocols.

Fault tolerance is achieved by executing an AOS application as multiple processes, possibly running on separate hardware devices connected by a network and separating the state from processing with support for replicated state and fast recovery of state.

AOS Architecture Overview

In the first part of this paper, we discussed some of the challenges to data center network architecture and how AOS works to solve those problems. Here, we delve into the specifics of AOS architecture.

AOS is based on distributed state-management infrastructure, which can be described as data-centric communication fabric with horizontally scalable and fault-tolerant in-memory data store. All the functionalities of the specific reference design application are implemented via a set of stateless agents. Agents communicate with each other via a logical publish-subscribe-based communication channel and essentially implement the application's logic.

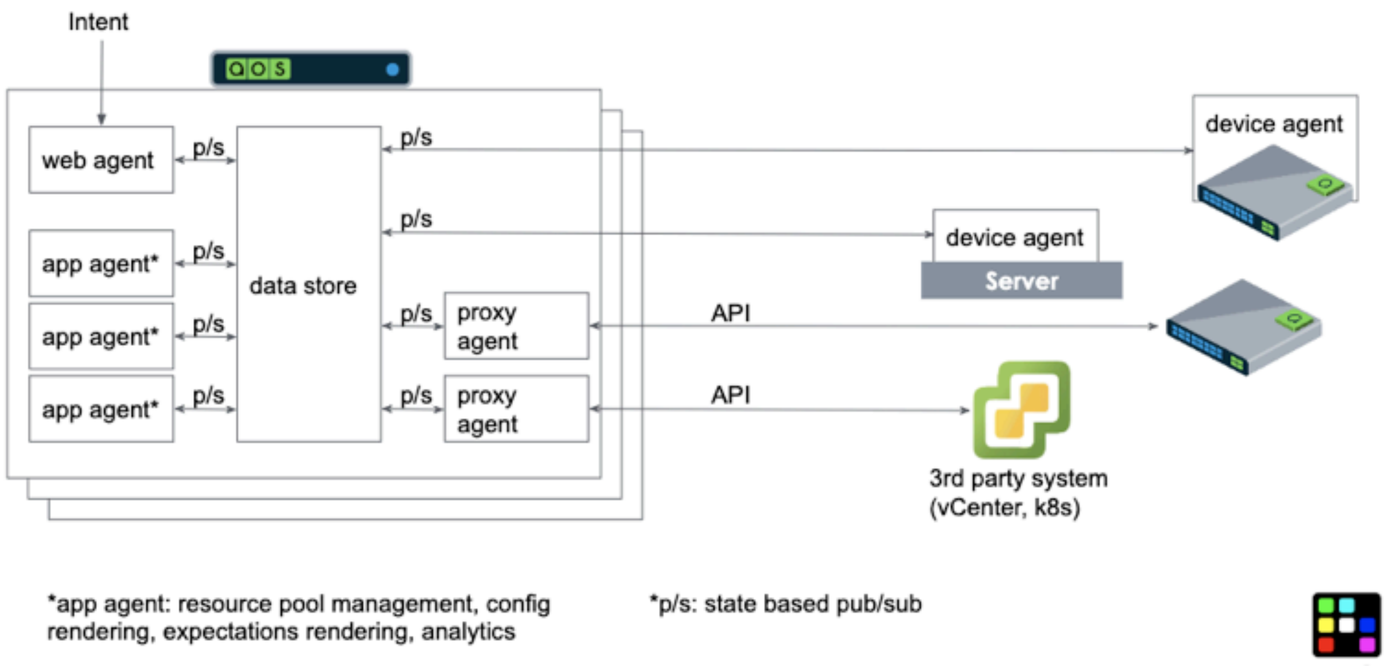
Every AOS reference design application is simply a collection of stateless agents described above. Broadly speaking, there are three classes of agents in AOS:

1. **Interaction (web) agents** are responsible for interacting with users, i.e., taking user input and feeding users with relevant context from the data store.

2. **Application agents** are responsible for performing application domain-specific data transformations, by subscribing to input entities and producing output entities.

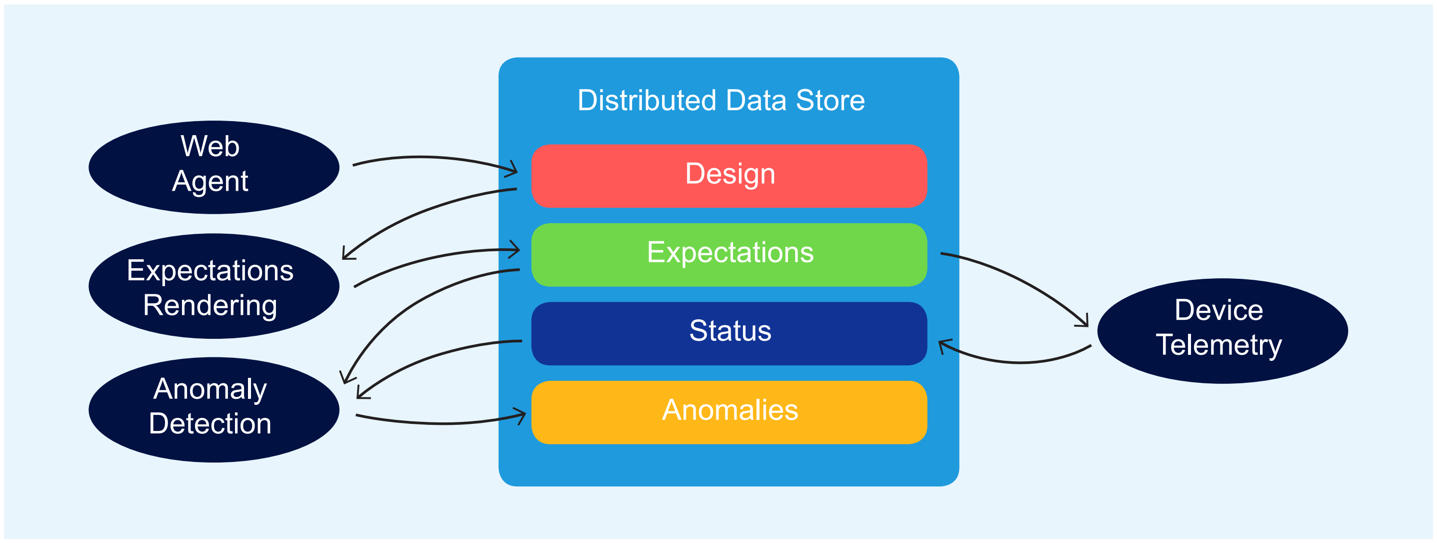
3. **Device agents** reside on (or are proxies for) a managed physical or virtual system such as a switch, server, firewall, load balancer or even controller and are used for writing configuration and gathering telemetry using native (device-specific) interfaces.

Figure 9: AOS Agents


Intent
AOS
web agent
app agent*
app agent*
app agent*
p/s
data store
proxy agent
proxy agent
API
device agent
device agent
Server
3rd party system (vCenter, k8s)
*app agent: resource pool management, config rendering, expectations rendering, analytics
*p/s: state based pub/sub

This interaction can be illustrated with an example describing a portion of the AOS data center networking reference design application.

Figure 10: AOS Data Center Networking Reference Design Application



The web agent takes user input — in this case, a design for an L3 Clos Fabric that contains the number of spines, leafs, and links between them, and the resource pools to use for fabric IPs and ASN numbers. The web agent publishes this intent into the data store as a set of graph nodes and relationships and their respective properties.

The build agent subscribes to this intent and:

- Performs correctness and completeness validations
- Allocates resources from resource pools

Then: Assuming the validations pass, the build agent publishes that intent along with resource allocations into the data store.

1. The config rendering agent subscribes to the output of the build agent.
2. For each node, the config agent fetches the relevant data, including resources, and merges it with configuration templates.
3. The expectations agent also subscribes to the output of the build agent and generates expectations that need to be met in order to validate the outcome.
4. The device telemetry agent subscribes to the output of the expectations agent and starts collecting relevant telemetry.
5. IBA probes process the raw telemetry and compare it against expectations and publish anomalies.
6. The RCI agent analyzes the anomalies and classifies them into symptoms, impacts and identified root causes.

Agents communicate via attribute-based interfaces (hence the term data-centric) by publishing entities and subscribing to changes in entities. Data-centric also implies that data definition is part of the framework and is implemented by defining the entities, as opposed to message-based systems, for example.

The data-centric publish-subscribe system does not suffer from the problems of message-based systems. In a message-based system, sooner or later the number of messages exceeds the capacity of the system to store or consume them. Dealing with this is hard as one has to replay the history of messages to get to a consistent state.

The data-centric system is resilient to surges in state changes as it is fundamentally dependent only on the last state. This state captures the important context and abstracts away all the possible (and irrelevant) event sequences that lead to it. Code written using state machine paradigm is easier to read, maintain, and debug, as described in depth in The Distributed Systems Challenge in Data Center Automation.

Hard problems (e.g., elasticity, fault tolerance) are solved once and on behalf of all agents. Typical architecture then consists of a number of stateless agents that can be restarted in case of failure and pick up where they left off by simply re-reading the state they subscribe to from sysdb.

Benefits of AOS Architecture

The AOS architecture provides significant benefits, solving some of the most difficult data center networking issues we've described in this paper. AOS architecture:

- **Helps operators deal with change reliably.** This is possible through real-time queryable intent and operational context
- **Simplifies all aspects of network service lifecycle**, including Day 0, 1, and 2 operations. And simplicity reduces the likelihood of operator error.
- **Reduces operational risk through stateful orchestration**, which leverages pre-condition validations, post-condition validations, automated config rendering and automated expectations validation

The Special Benefits of Root-Cause Identification and Intent-Based Analytics

The benefits of AOS's operational analytics components (root-cause identification and Intent-Based Analytics) include:

- **Reduce mean-time-to-repair and SME workload** through simplified operational analytics freeing your most-skilled resources to spend their time improving and innovating as opposed to dealing with fires
- **Extract more knowledge** by collecting and storing less data. Powered by the reference design behavioral contract, AOS knows what it is looking for and collects only that information. This on-the-fly processing can result in a 5-6 order of magnitude reduction in storage needs and post-processing of non-interesting data. This allows you to run your infrastructure more efficiently and remain competitive while controlling costs.
- **See issues quickly and easily**. AOS identifies significant, actionable events in a "simple pane of glass" and eliminates the noise of symptoms which are merely artifacts of an identified root cause.
- **Automate complex workflows**., AOS enables the automation of context-rich troubleshooting workflows (which otherwise is cumbersome, ineffective, time consuming and costly).
- **Enable zero-touch maintenance**. AOS has zero-touch, zero-cost maintenance in the presence of change as it is in constant sync with intent. As such it is resilient, automatically responds to changes and saves huge costs associated with maintaining data processing pipelines.
- **Eliminate costly DIY development**. DIY development of data-processing pipeline integration efforts are costly and fragile, takes time and resources away from your core business and requires a lot of SME heavy lifting.
- **Deliver highly accurate results** compared to machine learning/artificial intelligence approaches.
- **Employ a vendor-agnostic approach**, giving you freedom of choice between the best vendors and capabilities.
- **Use common APIs** across your public/private/hybrid cloud infrastructure.

Scale from Day 1: A Case Study

AOS is built with scaling in mind from day one. One customer used AOS to perform extensive validation.

- **Physical infrastructure** (6,000 interface status validations, 1,000 cabling validations, 36000 error counters, 10000 power, temperature, voltage metrics validations)
- **L2/L2 data plane** (12,000 queue drops counters, sanity of 100s of MLAGs, 3,000 STP validations, notifications of status changes)
- **Control plane** (1,500 BGP sessions health, ~500 expected next hops/default routes)
- **Capacity plan** (~500 trending analyses with configured thresholds for the usage of routing tables, ARP tables, multicast tables per VRF, 6,000 link usage validations)
- **Compliance** (ensuring expected OS versions running on all ~100 switches)
- **Multicast** (2,500 validations for expected PIM neighbors, 300 rendezvous point checks, 25 validations regarding detection of abnormal patterns in count of sources, groups, source-group pairs on RPs)

Essentially, instead of single pane of glass with 82,000 entries the customer was presented with a simple pane of glass that only shows anomalies, categorized into a dashboard per the customer's specification, which:

- Were 100% accurate (and not statistically inferred)
- Required zero-ongoing maintenance, since they were in constant sync with the customer's topology and intent
- Provided the relevant actionable context (what deviated, the nature of deviation, desired state)
- Saved on storage and processing needs (only 9GB RAM, 96GB disk required)
- Eliminated the danger of internal lock-in and legacy by writing complex and fragile data processing pipelines

Conclusion

The inability to make reliable changes in your IT infrastructure is a major obstacle to growth and innovation. AOS eliminates that fear and makes it possible to eradicate the dreaded "legacy infrastructure" forever, empowering you to make change reliably, which will in turn allow you to innovate reliably and stay competitive.