Quality of Service

thinqtank LEARNING

WHAT QOS ACTUALLY DOES

QoS prioritises some traffic to the detriment of other traffic. It creates no bandwidth. On an uncongested link it changes nothing at all; on a congested one it decides who suffers. Every tool below is an answer to a single question — *when there is not enough to go round, who waits and who is dropped?*

The four things it manages

	Is	Voice target
Bandwidth	how much can be sent per second	21–320 kbps per call, by codec
Delay (latency)	how long one packet takes to arrive	**≤ 150 ms** one way
Jitter	the *variation* in that delay	**≤ 30 ms**
Loss	packets that never arrive	**≤ 1 %**

Jitter is the one people underestimate. A call with a steady 140 ms delay sounds fine. The same call averaging 60 ms but swinging between 20 and 120 sounds broken — the jitter buffer runs dry and words drop out. Consistency beats speed.

Delay is cumulative and partly fixed: serialisation, propagation and processing happen whatever you configure. The only component QoS controls is **queuing delay**.

THREE MODELS

Model	How it works	Scales?
Best effort	no QoS at all. Every packet is equal and FIFO decides. The internet's default	infinitely
IntServ	RSVP signals and **reserves** a path end to end, per flow. A genuine guarantee — every router on the path must hold state for every call	**no**
DiffServ	mark the packet once; each hop applies its own **per-hop behaviour** to the marking. No reservation, no state, no guarantee	**yes**

Everything on this sheet is DiffServ. It wins not because it promises more but because it promises less: a router only has to read six bits and act, so it works at any scale and needs no memory of the flow.

THE TOOLKIT, IN ORDER

classify	mark	queue	avoid congestion	police / shape	link efficiency

Stage	Question it answers
Classification	what kind of traffic is this?
Marking	how do I record that, so the next hop does not have to work it out again?
Congestion management	the interface is full — what order do I send in?
Congestion avoidance	the queue is filling — what do I drop *before* it overflows?
Policing	this flow is over its rate — drop or re-mark it
Shaping	this flow is over its rate — buffer it and release it smoothly
Link efficiency	the link is slow — how do I stop one big packet blocking everything behind it?

Classify and mark once, at the edge. Act on the marking everywhere after that. Deep inspection at every hop is expensive and unnecessary — that is the whole design of DiffServ, and the reason the trust boundary matters as much as the queuing does.

1 — CLASSIFICATION

Deciding what a packet *is*. Done **as close to the source as possible**, because that is where the context still exists and the link is cheapest.

Tool	Matches on
Access lists	addresses, protocol, ports. Precise, and blind to anything that moves ports
NBAR / NBAR2	the application itself, by inspecting payload. Finds what a port number cannot
Existing markings	DSCP, IP precedence, CoS, MPLS EXP — trusting what an earlier hop already decided
Interface, MAC, protocol, VLAN	where it came from, rather than what it is
NetFlow	not a classifier in the data path — it tells you what is on the wire so you can design the policy

MQC — three steps, always the same three

```
R1(config)#class-map match-all VOICE
R1(config-cmap)#match ip dscp ef
```
1. What is it? `match-all` = every match statement must be true; `match-any` = any one will do
```
R1(config)#policy-map WAN-EDGE
R1(config-pmap)#class VOICE
R1(config-pmap-c)#priority percent 10
```
2. What do I do with it? mark, queue, police, shape, drop
```
R1(config)#interface GigabitEthernet0/0
R1(config-if)#service-policy output WAN-EDGE
```
3. Where does it apply? and in which direction

`class-default` exists whether you write it or not, and catches everything unmatched. Configure it deliberately — left alone it is plain FIFO.

THE TRUST BOUNDARY

A marking is only worth acting on if you believe it. The **trust boundary** is the line past which markings are accepted rather than rewritten — and it should sit as far toward the edge as the devices there can be trusted.

At the access port	Do
A PC or printer	**do not trust.** Any application can set DSCP on its own packets
An IP phone	**trust the phone**, and have the phone rewrite the PC behind it to 0
An AP, camera, or other managed device	trust conditionally, once you know what it marks
A switch or router you administer	trust

```
Switch(config-if)#mls qos trust device cisco-phone
Switch(config-if)#mls qos trust cos
```
Classic Catalyst IOS — conditional trust: honour CoS *only* while CDP or LLDP confirms a Cisco phone is on the port. Unplug the phone and trust drops away
```
Switch(config-if)#trust device cisco-phone
```
IOS-XE Catalyst 9000. There is no `mls qos` global switch on these — QoS is always on

Untrusted does not mean unmarked. An untrusted port rewrites incoming markings to 0 by default. If you leave a marking you did not authorise in place, a workstation can put its own traffic in your voice queue — and it will get there ahead of the phones.

2 — MARKING AT LAYER 2

CoS — Class of Service, three bits of the Priority Code Point field inside the 802.1Q tag. Eight values, 0 to 7.

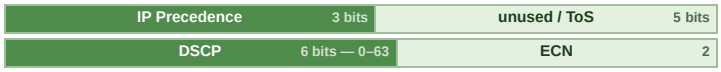
CoS lives in the tag, so no tag means no CoS. An ordinary access port sends untagged frames and carries no CoS at all. The marking survives exactly as far as the next 802.1Q trunk and is stripped the moment the frame is routed — which is why anything that must survive end to end has to be marked at Layer 3.

CoS	Conventional use
7, 6	reserved for network control — do not hand these out
5	**voice**
4	video
3	**call signalling**
2, 1	data classes
0	best effort — the default

The same three-bit idea appears as **MPLS EXP** in a label, **UP** in 802.11 WMM, and **DE** or **CLP** as a single drop-eligible bit in Frame Relay and ATM. Different names, same job at the link layer.

MARKING AT LAYER 3

The IPv4 header's second byte — the **ToS byte**, and the IPv6 **Traffic Class** byte — has been redefined once. Both readings are still in use, which is why the same field has two names.



The two are deliberately compatible: **the first three bits of a DSCP value are the old IP precedence**. That is what the Class Selector values exist to preserve.

The three per-hop behaviours

PHB	Means
Default (DF) `DSCP 0`	best effort. Anything unmarked
Class Selector `CS1–CS7`	the last three bits are zero, so the value *is* an IP precedence. Backwards compatibility, and still used for control and signalling
Assured Forwarding `AF11–AF43`	four classes, each with three drop precedences. A guarantee of bandwidth, not of delay
Expedited Forwarding `EF = 46`	**low loss, low latency, low jitter.** One value, and it belongs to voice

THE DSCP VALUES WORTH KNOWING

Class Selector — `CSn = n × 8`

CS1	CS2	CS3	CS4	CS5	CS6	CS7
8	16	24	32	40	48	56

Assured Forwarding — `AFxy = 8x + 2y`

	Low drop y = 1	Medium drop y = 2	High drop y = 3
AF4*y* x = 4	**34**	36	38
AF3*y* x = 3	**26**	28	30
AF2*y* x = 2	**18**	20	22
AF1*y* x = 1	**10**	12	14

x is the class and decides the queue. **y** is the drop precedence and decides who is thrown out of that queue first when it fills — higher **y** is dropped sooner. AF11 and AF13 sit in the same queue; AF13 dies first.

Expedited Forwarding

`EF = 46 = 101110`

First three bits `101` = IP precedence 5. One value, one job: voice.

Two you can work out in your head. EF is 46 and **CS*n* is n×8**. From those two, plus `AFxy = 8x + 2y`, every value on this page is derivable without the table.

A WORKING CLASS MODEL

Cisco's twelve-class model, after RFC 4594. Few networks need all twelve — the point is that whatever you use should be a **subset of this**, so it still means the same thing to the carrier and to the next network.

Traffic class	DSCP	CoS	Queue
Network control	CS6 48	6	bandwidth
VoIP telephony	**EF 46**	**5**	**priority**
Broadcast video	CS5 40	5	priority
Real-time interactive	CS4 32	4	priority
Multimedia conferencing	AF4*y*	4	bandwidth + WRED
Multimedia streaming	AF3*y*	4	bandwidth + WRED
Call signalling	**CS3 24**	**3**	bandwidth
Network management	CS2 16	2	bandwidth
Transactional data	AF2*y*	2	bandwidth + WRED
Bulk data	AF1*y*	1	bandwidth + WRED
Scavenger	**CS1 8**	1	minimum
Best effort	DF 0	0	default + WRED

Scavenger is the useful oddity. CS1 is marked *worse* than unmarked traffic on purpose — a class that gets whatever is left and nothing more. Backups, software distribution and recreational traffic go here, and on a congested link they get out of the way without ever being blocked outright.

```
R1(config-pmap-c)#set dscp ef
R1(config-pmap-c)#set cos 5
```

Mark once, at the edge, in both layers where the frame will cross a trunk

3 — CONGESTION MANAGEMENT

Queuing only happens when the interface is **already full**. An uncongested link has an empty queue and every scheme below behaves identically — which is why a policy that "does nothing" is usually a policy on a link that is not congested yet.

Scheme	How it serves	Weakness
FIFO	one queue, first in first out. The default on fast interfaces	no prioritisation at all
PQ	four fixed queues; the higher one is always emptied first	**starves the lower queues**
CQ	sixteen queues, round robin, a byte count each	no queue ever gets strict priority, so no low-delay class
WFQ	automatic per-flow fairness. Default on links at or below 2.048 Mbps	flows, not classes — you cannot name what matters
CBWFQ	your own classes, each with a **guaranteed minimum** bandwidth	a minimum is not low delay
LLQ	**CBWFQ plus one strict-priority queue, policed**	none worth mentioning — this is the answer

Why LLQ exists. Priority queuing gives voice the low delay it needs and starves everything else. CBWFQ shares fairly but a bandwidth guarantee says nothing about *when* you are served. LLQ bolts a strict-priority queue onto CBWFQ and **polices it**: voice is always served first, but only up to its allowance — past that it is dropped rather than allowed to starve the rest.

```
R1(config-pmap)#class VOICE
R1(config-pmap-c)#priority percent 10
     The strict-priority queue, with its policer. This is the LLQ part
R1(config-pmap)#class SIGNALLING
R1(config-pmap-c)#bandwidth percent 5
     A guaranteed minimum — available above it when nobody else wants it
R1(config-pmap)#class class-default
R1(config-pmap-c)#fair-queue
     WFQ inside the leftover class, so no single flow dominates it
```

Keep the priority queue at or below a third of the link — past that the rest of the policy stops meaning anything. Only **75 %** is reservable by default; the remainder is left for control traffic and overhead, and `max-reserved-bandwidth` should rarely be touched.

4 — CONGESTION AVOIDANCE

Queuing decides the order of what is sent. Congestion avoidance decides **what to throw away before the queue is full** — and the reason to bother is what happens if you do not.

Tail drop, and why it is worse than it looks

Once the queue is full, drop whatever arrives. But on a link carrying many TCP sessions they all lose packets in the same instant, back off together and ramp up together. Throughput collapses and recovers in waves, and the link is never actually full. That is **TCP global synchronisation**.

WRED

Start dropping **early and at random**, so a few sessions back off at a time instead of all of them. Weighted, so the drops are aimed: higher drop precedence goes first.

Knob	Effect
Minimum threshold	queue depth at which dropping begins
Maximum threshold	above this, drop everything — back to tail drop
Mark probability denominator	at the maximum threshold, one in *n* is dropped. Lower *n*, more aggressive

```
R1(config-pmap-c)#random-detect dscp-based
     Aim the drops by DSCP, so AF13 is discarded well before AF11
R1(config-pmap-c)#random-detect dscp af13 24 40 10
     min 24, max 40, drop one in 10 at the maximum
```

WRED only works on TCP. The mechanism depends on the sender noticing a drop and slowing down. UDP does not, so dropping a voice packet early achieves nothing except a worse call. **Never apply WRED to a priority queue. ECN** is the same idea without the loss — set the two ECN bits instead of dropping and let the receiver ask the sender to slow down, where both endpoints support it.

5 AND 6 — POLICING AND SHAPING

Both limit a flow to a rate. They differ in one decision, and everything else follows from it: **what happens to the excess.**

	Policing	Shaping
Excess traffic is	**dropped or re-marked**	**buffered and delayed**
Buffers	no	yes — which costs memory and adds delay
Direction	inbound **or** outbound	outbound only
Output rate	bursty — the shape is passed through	smoothed
Effect on TCP	retransmissions, and sawtooth throughput	senders adapt cleanly
Typical use	enforce a rate on ingress, cap a class, re-mark to scavenger	match a carrier's CIR so the carrier never has to drop

The classic case for shaping. You buy 50 Mbps on a gigabit handoff. Send at line rate and the carrier polices you — and it drops without regard for your markings, so voice goes with everything else. Shape to 50 Mbps yourself and the queuing you configured decides what waits, in your own priority order. **Shape so that you, not the carrier, choose what suffers.**

The token bucket

CIR	committed information rate — the rate you are entitled to
Bc	committed burst — tokens added each interval
Be	excess burst — credit for unused capacity
Tc	the interval, and it is derived: `Tc = Bc / CIR`

Two colours, or three

Model	Outcomes
Single rate, two colour	**conform** → transmit · **exceed** → drop or re-mark
Single rate, three colour (srTCM)	CIR with Bc and Be: **conform** · **exceed** · **violate**
Two rate, three colour (trTCM)	**CIR and PIR.** Under CIR conform, between CIR and PIR exceed, above PIR violate

```
R1(config-pmap-c)#police cir 5000000 pir 10000000
R1(config-pmap-c-police)#conform-action transmit
R1(config-pmap-c-police)#exceed-action set-dscp-transmit cs1
R1(config-pmap-c-police)#violate-action drop
     Two rate, three colour. Note the middle action: rather than dropping, demote the traffic
     to scavenger and let it through if there is room
R1(config-pmap-c)#shape average 50000000
     Shape the whole class to 50 Mbps, then nest the queuing policy inside it with
     service-policy
```

Re-marking beats dropping wherever the traffic is not urgent. A dropped TCP segment is retransmitted and costs the link twice; a demoted one is carried when there is capacity and quietly loses when there is not.

7 — LINK EFFICIENCY

Tools for links slow enough that the **size of a packet** is itself a delay. On anything above roughly 768 kbps they are unnecessary; below it they are the difference between usable voice and none.

Header compression

A G.729 voice packet carries 20 bytes of payload under **40 bytes** of IP, UDP and RTP header — twice as much header as data. **cRTP** compresses that 40 down to 2–5 bytes by sending only what changed between packets.

Roughly halves voice bandwidth	on the links where bandwidth is scarcest
Costs CPU on both ends	it is hop by hop, so both routers must run it
Slow links only	below about 2 Mbps. Above that the CPU costs more than the bandwidth saves

Payload compression

STAC, Predictor and MPPC compress the data itself rather than the header. Rarely worth it now — most traffic is already compressed or encrypted, and neither compresses again.

Fragmentation and interleaving

A 1500-byte packet takes **214 ms** to clock onto a 56 kbps link. A voice packet arriving behind it waits the whole time — and the budget for the entire one-way path is 150 ms. No amount of priority queuing helps: the big packet is already on the wire and cannot be interrupted.

LFI chops the large packet into pieces and interleaves voice between them. Size the fragments for a serialisation delay of **10 to 15 ms**:

```
fragment size (bytes) = link speed (bps) × 0.01 ÷ 8
```

56 kbps → 70 bytes · 128 kbps → 160 bytes · 768 kbps → 960 bytes

Implemented as **MLP LFI** on PPP links and **FRF.12** on Frame Relay. Both are legacy technologies, and so is the problem — but the arithmetic is the clearest illustration on the sheet of why a queue is not always the answer.

VERIFICATION

Command	Shows
`show policy-map interface int`	**the one that matters** — per class, packets matched, bytes, drops, queue depth and policer counts
`show policy-map`	the policy as configured
`show class-map`	what each class matches on
`show mls qos interface int`	trust state and queueing on a Catalyst port
`show queue int`	what is sitting in the queue right now
`show interfaces int`	output drops — the number that tells you congestion is real

Read the counters, not the configuration. A class matching zero packets is the commonest QoS fault by a distance, and it looks perfect in `show running-config`. Only `show policy-map interface` reveals it — usually a marking that never arrived, an ACL pointing the wrong way, or a policy applied in the wrong direction.

The order to check

1. **Is the link actually congested?** No output drops means QoS has nothing to do, and the problem is elsewhere.
2. **Are packets matching the classes?** If not, it is a classification or marking fault, not a queuing one.
3. **Is the marking surviving?** Check it at each hop. A trust boundary in the wrong place rewrites to 0 and everything downstream behaves correctly on the wrong value.
4. **Is the policy on the right interface and direction?** Shaping and queuing are outbound; the congested direction is the one that needs it.

WHERE QOS GOES WRONG

Symptom	Nearly always
Policy configured, nothing improves	the link is not congested. No output drops means QoS is idle. Look elsewhere
A class matches zero packets	the marking never arrived, or the ACL is inverted, or the policy is on the wrong direction
Voice fine on the LAN, bad over the WAN	CoS was set but not DSCP. The tag is stripped the moment the frame is routed
Everything is marked EF	a trust boundary in the wrong place. If all traffic is priority, none of it is
Voice degrades under heavy voice load	the LLQ policer is doing its job. The priority queue is full and dropping its own class
Carrier drops your traffic despite the policy	you are policed at their edge. Shape to their CIR on your side
Marking correct at one hop, gone at the next	an untrusted port between them rewrote it to 0

The one that costs the most time. QoS is only ever visible during congestion. A policy tested on a quiet link tells you nothing about whether it works — and a policy that has never been tested under load is a policy nobody has tested.

DEPLOYING IT

1. **Find out what is actually on the wire.** NetFlow or NBAR first, policy second. A class model designed from assumptions protects traffic nobody sends.
2. **Agree the classes, and keep them few.** Four or five classes that everyone understands beat twelve that drift out of step between sites.
3. **Set the trust boundary at the access edge** and mark there. Every hop after it acts on the marking rather than re-deriving it.
4. **Mark at Layer 3.** CoS where a trunk needs it, DSCP always — DSCP is the one that survives being routed.
5. **Apply queuing where congestion actually is** — the WAN edge and any speed mismatch. A policy on an uncongested core link does nothing.
6. **Shape to the rate you bought**, and nest the queuing inside the shaper so your priorities, not the carrier's drops, decide what waits.
7. **Verify under load, with counters.** `show policy-map interface` during a busy period is the only proof that any of it works.

The end-to-end rule. QoS is per hop, and a chain of hops is only as good as the worst one. One switch in the middle with the wrong trust setting undoes every other device on the path — and it will look perfectly configured, because it is doing exactly what it was told.