Gateway Executor Performance — Comprehensive Load Test Report

---

1. Introduction & Objective

This report presents the results of a comprehensive, automated load test of the **WSO2 API Platform Gateway**, specifically targeting the performance characteristics of the **Python policy executor** compared to the native **Go policy engine**.

The key question this test answers is: **What is the measurable performance cost of the Python executor compared to native Go, across different dimensions of load?**

> **Important**
>
> We are **not** testing the logic of any specific policy. We are testing the **executor infrastructure** — the overhead of invoking a policy, serializing/deserializing payloads across the gRPC bridge, and maintaining stability under load. This is why we use intentionally minimal "stub" policies.

---

2. Test Policies — What Was Executed

2.1 Why Minimal Policies?

To isolate executor overhead from policy logic, we use **minimal paired sample policies**. Each policy performs the absolute minimum work possible for its hook type — setting a single header or prefixing a body. The Go and Python variants are **intentionally matched** so they perform identical operations, making any latency difference attributable purely to the executor runtime.

2.2 Policy Catalog

There are **8 minimal policies**, 4 in Go and 4 in Python:

Policy Name	Language	Hook	What It Does
`sample-go-request-header`	Go	Request Header	Sets `x-sample-policy-phase: request-header`
`sample-go-response-header`	Go	Response Header	Sets `x-sample-policy-phase: response-header`

Policy Name	Language	Hook	What It Does
`sample-go-request-body`	Go	Request Body	Prefixes body with `[sample-request-body]`
`sample-go-response-body`	Go	Response Body	Prefixes body with `[sample-response-body]`
`sample-python-request-header`	Python	Request Header	Sets `x-sample-policy-phase: request-header`
`sample-python-response-header`	Python	Response Header	Sets `x-sample-policy-phase: response-header`
`sample-python-request-body`	Python	Request Body	Prefixes body with `[sample-request-body]`
`sample-python-response-body`	Python	Response Body	Prefixes body with `[sample-response-body]`

2.3 Test Profiles (Policy Chains)

Since the netty echo backend does not echo request headers back in the response, we avoid `request-header` policies and instead duplicate `response-header` to maintain a 4-hook chain:

Profile	Policies in Chain	Purpose
No Gateway	*(JMeter → Netty directly)*	Pure backend latency — bypasses gateway entirely
No Policies	*(none)*	Gateway overhead — measures Router + Envoy + upstream
Go 4-Policy	go-resp-header → go-req-body → go-resp-header → go-resp-body	All Go hooks active
Python 4-Policy	py-resp-header → py-req-body → py-resp-header → py-resp-body	All Python hooks via gRPC bridge
Mix 4-Policy	go-resp-header → py-req-body → go-resp-body → py-resp-header	Interleaved Go+Python chain

2.4 Per-Request Validation

Every single request is validated by a **JMeter Groovy assertion** that checks:

- Policy profiles: response header `x-sample-policy-phase: response-header` is present, body contains `[sample-response-body]` and `[sample-request-body]` prefixes
- No-policy profiles: no policy headers or body mutations present
- Original payload preserved in all cases

Result: 100% accuracy across all 25 tests — every request produced correct policy mutations.

3. Test Environment

3.1 Docker Host Machine

Property	Value
OS	macOS 13.7.8 (x86_64)
Available CPUs	2 cores (allocated to Docker)
Available Memory	~3.83 GB (allocated to Docker)
Docker Engine	Docker Desktop for Mac

3.2 Gateway Controller Container

Property	Value
Container	`executor-benchmark-gateway-controller-1`
Image	`ghcr.io/wso2/api-platform/gateway-controller:1.1.0-SNAPSHOT`
CPU Limit	**0.025 cores** (25 millicores)
Memory Limit	**60 MB** (hard limit)
Storage	SQLite (development mode)
Ports	REST API `:9090`, Admin `:9094`, xDS gRPC `:18000/:18001`

3.3 Gateway Runtime Container

Property	Value
Container	`executor-benchmark-gateway-runtime-1`
Image	`ghcr.io/wso2/api-platform/gateway-runtime:1.1.0-SNAPSHOT`
CPU Limit	**Uncapped** (inherits full Docker host allocation = 2 CPUs)
Memory Limit	**Uncapped** (~3.83 GB available from Docker VM)
Components	Envoy Router + Go Policy Engine + Embedded Python Runtime
Ports	HTTP ingress `:8080`, HTTPS `:8443`, Admin `:9002`, Metrics `:9003`
Environment Variables	`GOMAXPROCS`, `ROUTER_CONCURRENCY`, `PYTHON_POLICY_WORKERS`, `PYTHON_POLICY_MAX_CONCURRENT`, `PYTHON_POLICY_TIMEOUT` — all set to system defaults

3.4 Backend

Property	Value
Container	`executor-benchmark-netty-1`
Image	`renukafernando/netty-http-echo-service:0.4.6`
Description	HTTP echo service — returns request body as response body
Port	`:8688`

> **Note**
>
> The runtime container runs **without** any CPU or memory caps to measure the true performance ceiling. The gateway environment is completely restarted (docker compose down -v && up) between each of the 25 tests for a clean state.

---

4. Test Methodology

4.1 Test Matrix

25 tests = 5 profiles × (4 concurrency levels at 50B + 1 payload size at c=1)

#	Payload	Concurrency	Tests
1–5	50B	1	All 5 profiles
6–10	50B	2	All 5 profiles
11–15	50B	5	All 5 profiles
16–20	50B	10	All 5 profiles
21–25	1KiB	1	All 5 profiles

4.2 Per-Test Lifecycle

Each test executes this sequence:

Step	What It Does	Duration
1. Clean Restart	`docker compose down -v` + `up -d`	~15s
2. Wait for Services	Poll controller + runtime health endpoints	~5s
3. Create API	POST to controller REST API with policy chain (skipped for no-gateway)	~3s
4. Wait for Route	Poll gateway until route returns HTTP 200	~5s

Step	What It Does	Duration
5. Start Resource Monitor	Background `docker stats` capture every 20 seconds	continuous
6. Run JMeter	6-minute load test (1 min warmup + 5 min measured)	360s
7. Capture Results	Save JTL, resource stats, test metadata	~1s
8. Teardown	Stop monitor, compose down, 10s cooldown	~15s

4.3 What Was Measured

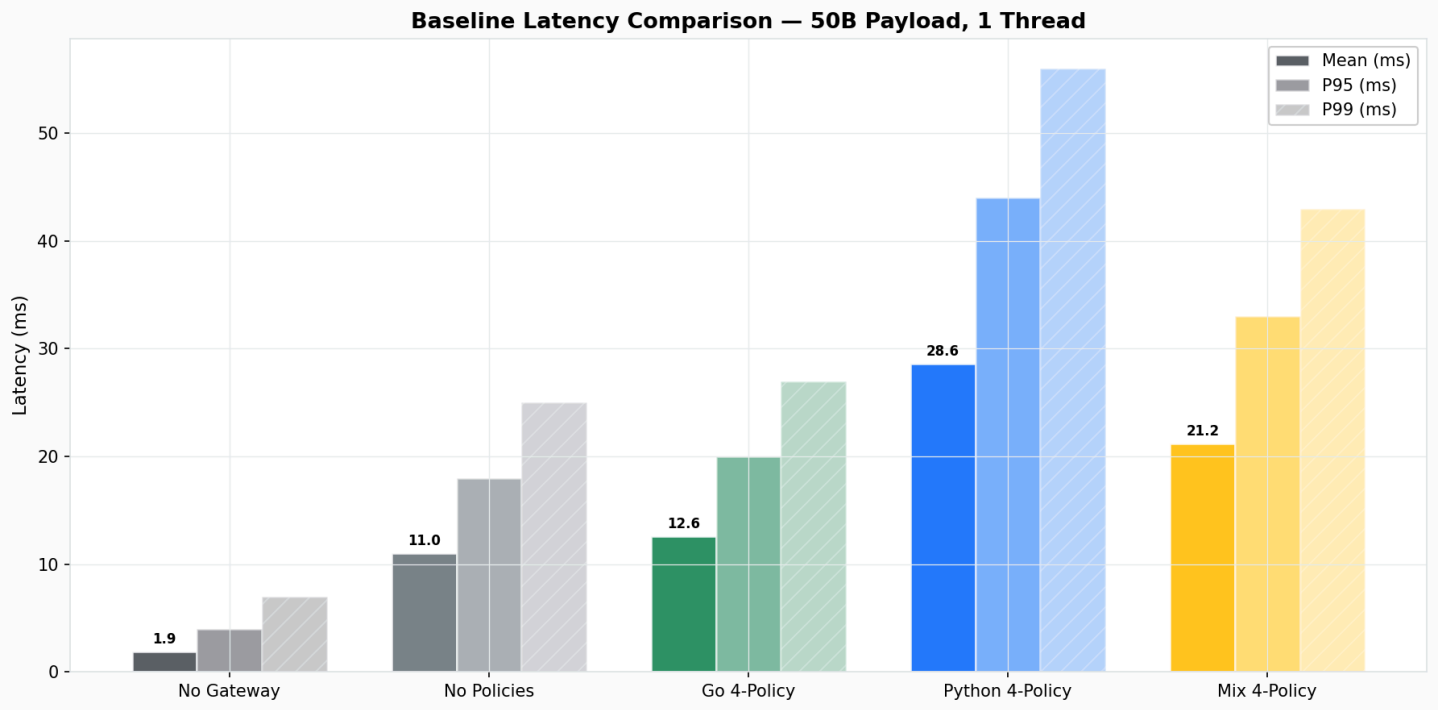
- **Request metrics** (from JTL): Response time (mean, median, P95, P99, max), throughput (TPS), error rate, accuracy
- **Container resources** (via `docker stats` every 20s): CPU %, memory usage, PID count
- **Warmup filtering**: First 60 seconds of each test are discarded from latency calculations

---

5. Results

5.1 Baseline Latency — Single Thread Performance

This is the most important measurement for understanding raw executor overhead. Each profile was tested with **1 thread** and **50B payload**, eliminating queuing effects.



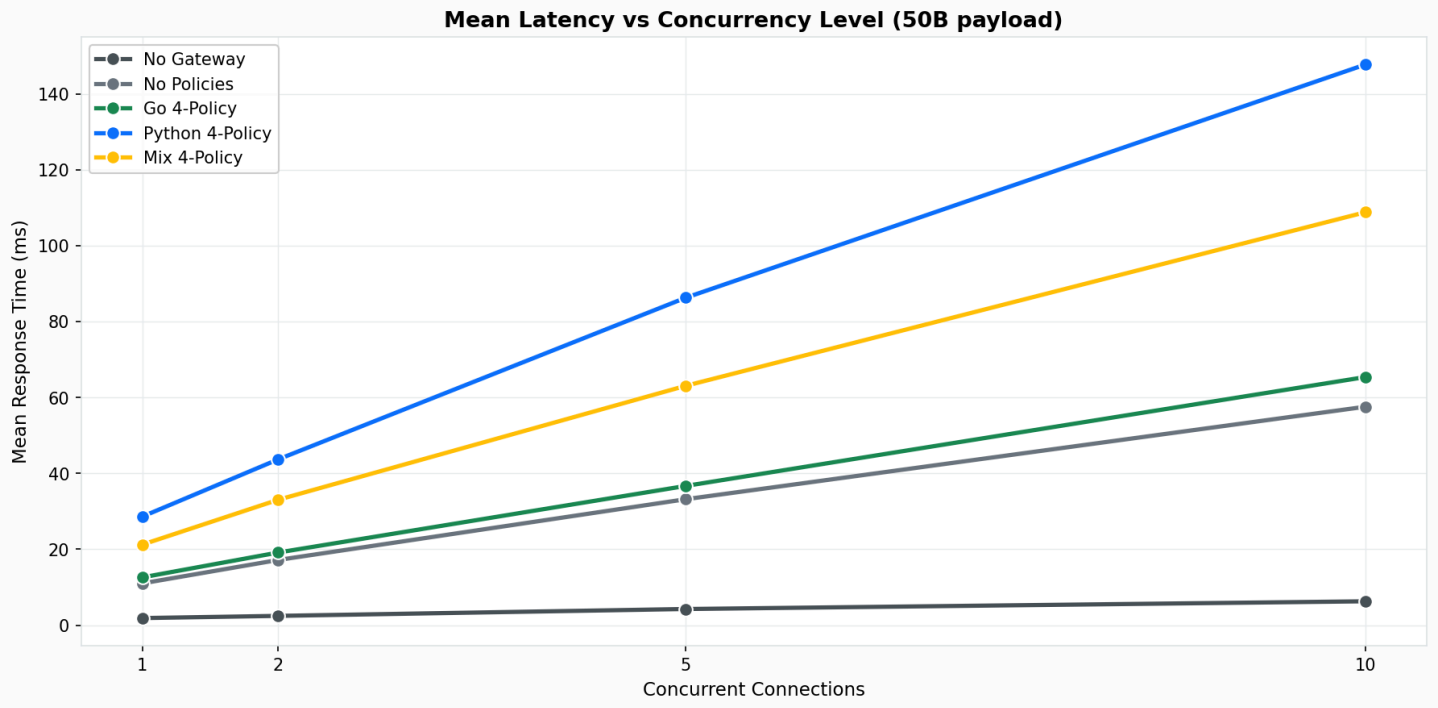
Profile	Mean (ms)	Median (ms)	P95 (ms)	P99 (ms)	Max (ms)	TPS
No Gateway	1.87	2	4	7	178	499.80
No Policies	11.01	10	18	25	90	88.91

Profile	Mean (ms)	Median (ms)	P95 (ms)	P99 (ms)	Max (ms)	TPS
Go 4-Policy	12.59	12	20	27	73	78.02
Python 4-Policy	28.60	27	44	56	95	34.60
Mix 4-Policy	21.22	20	33	43	72	46.52

> **Tip**
>
> **Key Finding:** The Python 4-policy chain adds **~17.6ms** overhead compared to the no-policies baseline (28.6ms - 11.0ms). The Go 4-policy chain adds only **~1.6ms**. Per individual hook, the Python gRPC bridge costs approximately **4.4ms** of additional latency. The Go engine adds only **~0.4ms** per hook — almost free.

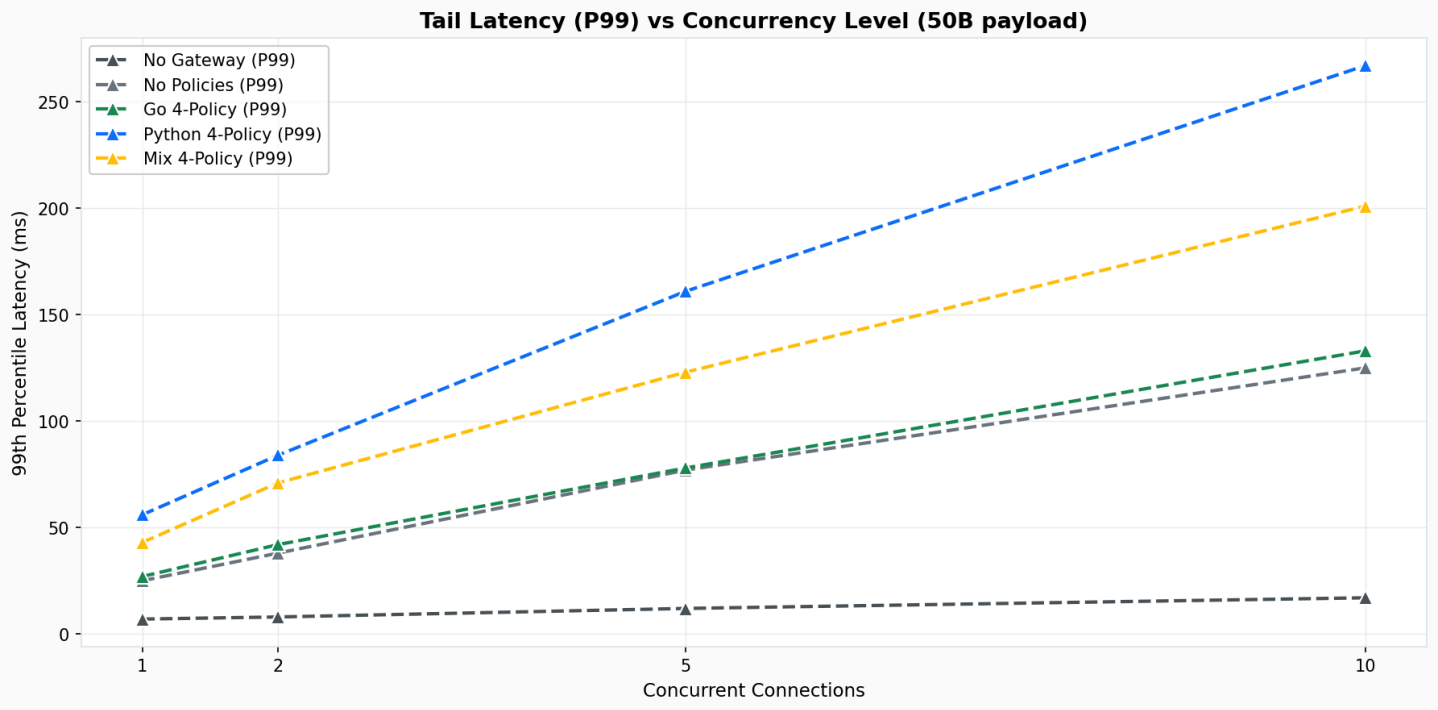
5.2 Concurrency Scaling — Latency Under Load

Each concurrency level runs for 5 minutes (after warmup) across all profiles.



Concurrency	No Gateway (ms)	No Policies (ms)	Go 4-Policy (ms)	Python 4-Policy (ms)	Mix 4-Policy (ms)
1	1.87	11.01	12.59	28.60	21.22
2	2.45	17.18	19.14	43.68	33.04
5	4.26	33.19	36.69	86.28	63.07
10	6.29	57.55	65.37	147.70	108.79

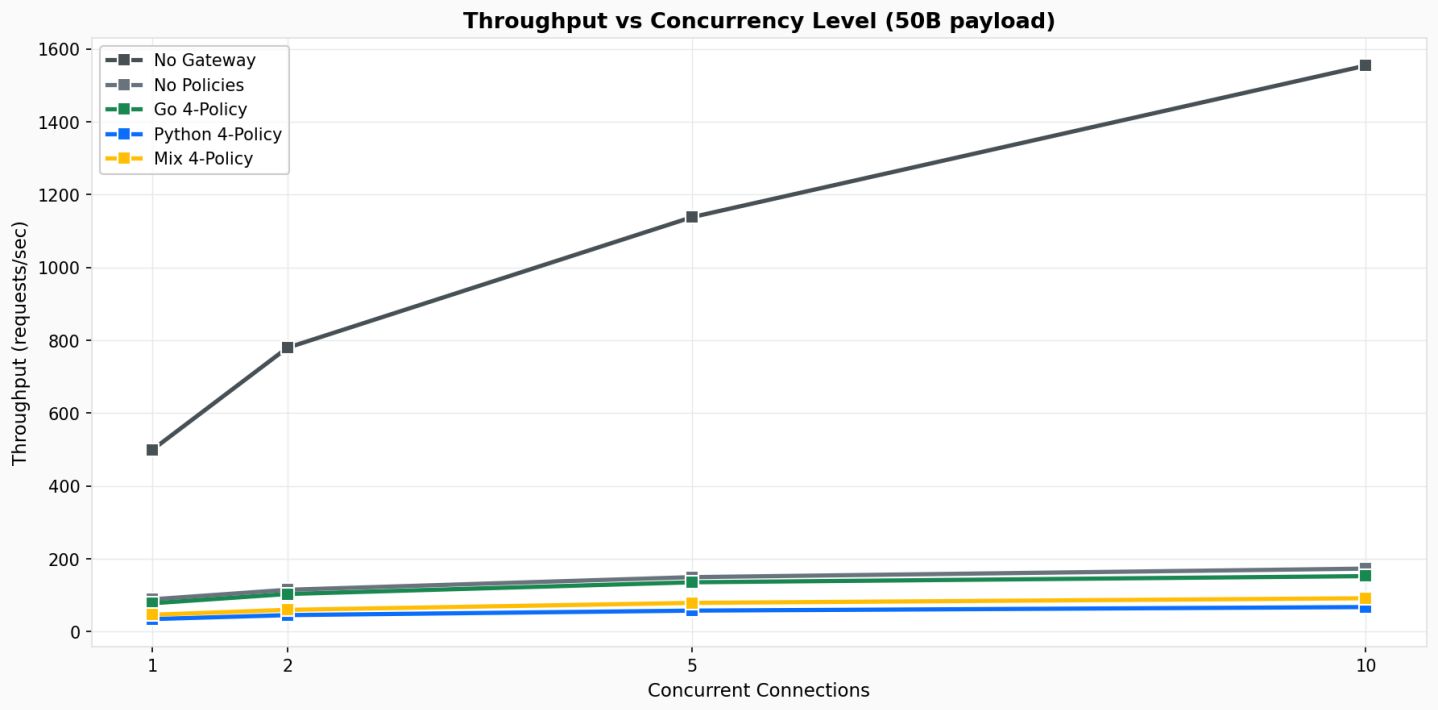
5.3 Tail Latency (99th Percentile)



Concurrency	No Policies P99 (ms)	Go 4-Policy P99 (ms)	Python 4-Policy P99 (ms)	Mix 4-Policy P99 (ms)
1	25	27	56	43
2	38	42	84	71
5	77	78	161	123
10	125	133	267	201

> **Note**
>
> **Key Finding:** Python P99 latency is approximately **2x** the no-policies baseline at every concurrency level. This ratio remains remarkably stable — it does **not** worsen under load, indicating the Python executor scales linearly and does not introduce contention or bottlenecks.

5.4 Concurrency Scaling — Throughput

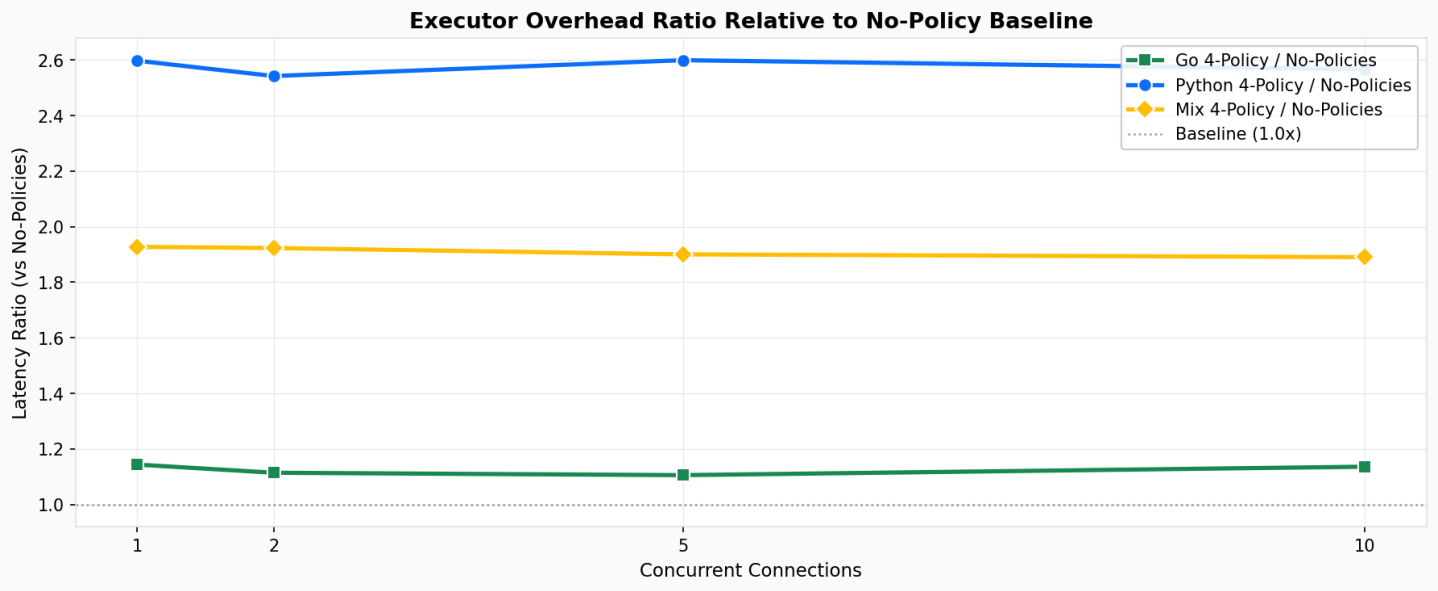


Concurrency	No Gateway (TPS)	No Policies (TPS)	Go 4-Policy (TPS)	Python 4-Policy (TPS)	Mix 4-Policy (TPS)
1	499.80	88.91	78.02	34.60	46.52
2	779.64	115.02	103.43	45.47	59.95
5	1,138.17	149.74	135.62	57.79	79.02
10	1,554.60	173.17	152.55	67.58	91.75

> **Tip**
>
> **Key Finding:** The no-gateway baseline achieves **1,554 TPS** at concurrency 10 — demonstrating the netty backend is not the bottleneck. Through the gateway, throughput tops out at **173 TPS** for no-policies, showing the Envoy+Router overhead dominates. The Go 4-policy chain achieves **88%** of the no-policy throughput. The Python 4-policy chain achieves **39%** of the no-policy throughput at c=10.

5.5 Executor Overhead Ratio

This graph shows the latency multiplier of each executor profile relative to the no-policies baseline:

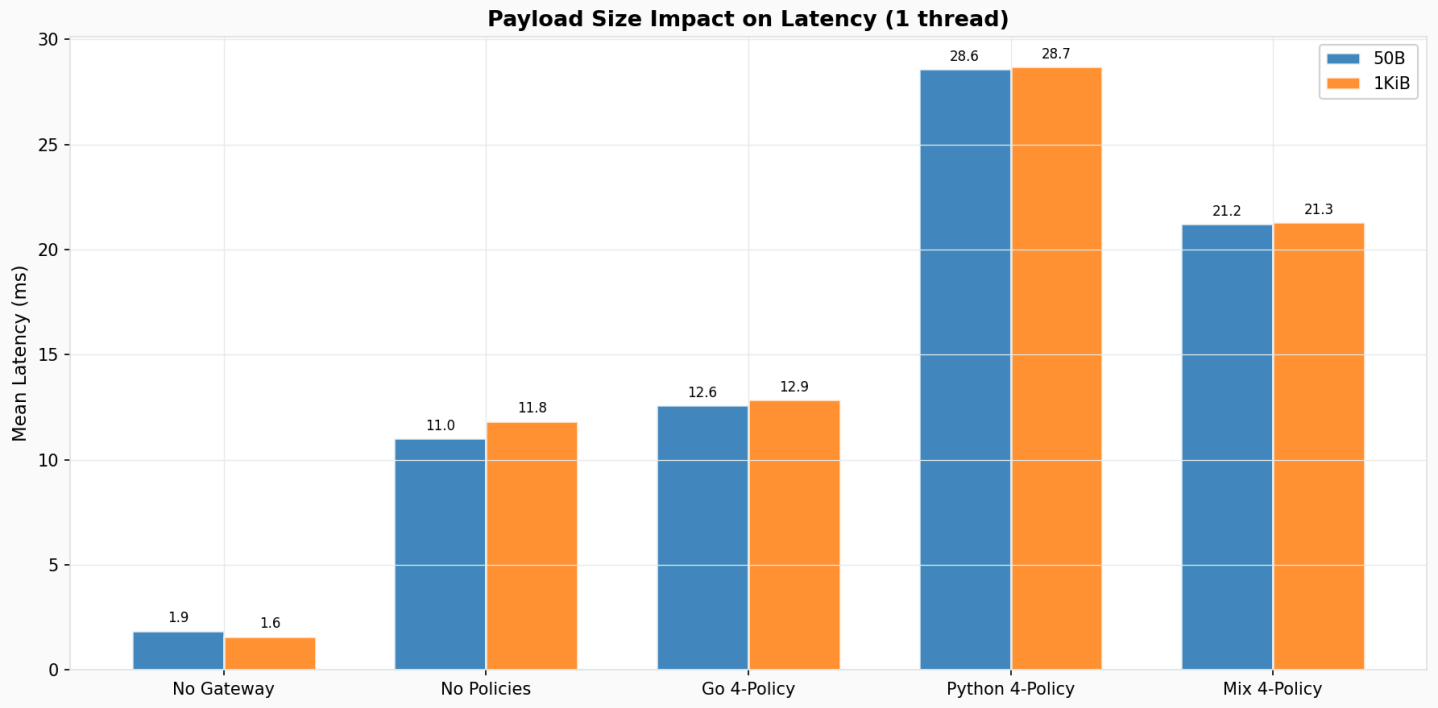


Concurrency	Go 4-Policy / No-Policies	Python 4-Policy / No-Policies	Mix 4-Policy / No-Policies
1	1.14x	2.60x	1.93x
2	1.11x	2.54x	1.92x
5	1.11x	2.60x	1.90x
10	1.14x	2.57x	1.89x

> **Important**
>
> **Key Finding:** The overhead ratios are **remarkably flat** across all concurrency levels. The Go engine adds a constant **~1.1x** overhead. The Python executor adds a constant **~2.6x** overhead. The Mix profile (2 Go + 2 Python) sits at **~1.9x**. None of these ratios degrade with increased load — the executors scale proportionally.

5.6 Payload Size Impact

This test compares 50B vs 1KiB payloads at concurrency 1, measuring serialization cost:

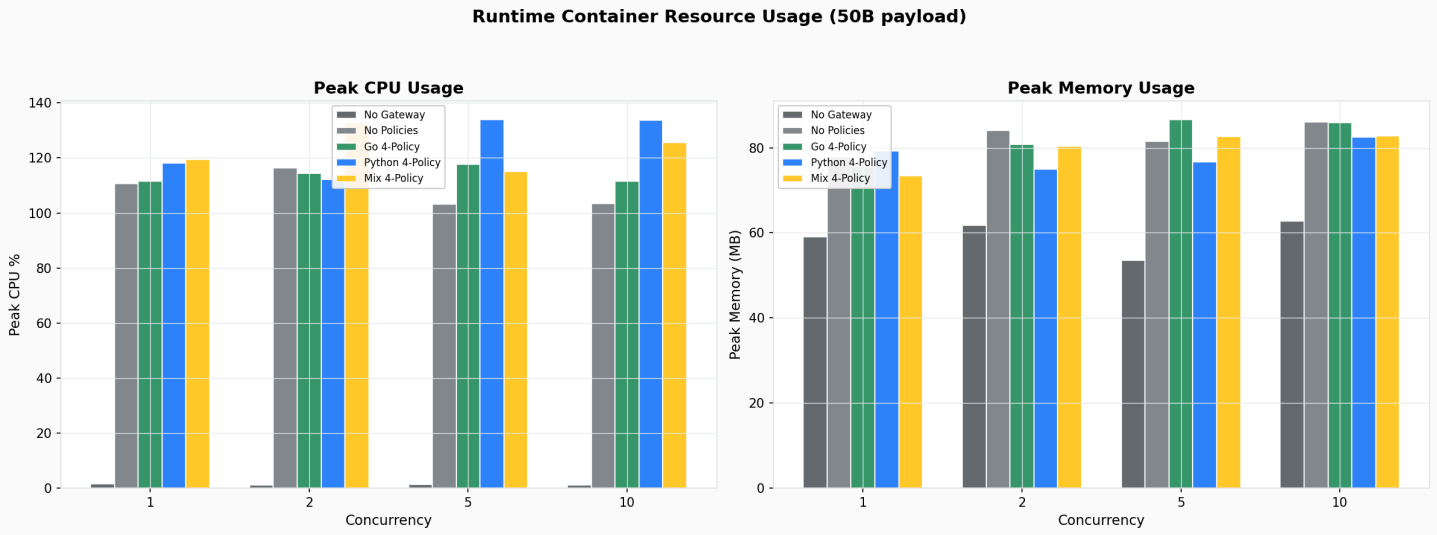


Profile	50B Mean (ms)	1KiB Mean (ms)	Increase (ms)	Increase (%)
No Gateway	1.87	1.60	-0.27	-14.4%
No Policies	11.01	11.84	+0.83	+7.5%
Go 4-Policy	12.59	12.87	+0.28	+2.2%
Python 4-Policy	28.60	28.71	+0.11	+0.4%
Mix 4-Policy	21.22	21.31	+0.09	+0.4%

Note

Key Finding: The payload size increase from 50B to 1KiB has **negligible impact** on latency across all profiles. The Python gRPC bridge serialization cost is effectively zero for payloads at this scale. The 50B → 1KiB jump adds less than 1ms even for the Python executor, indicating the protobuf serialization overhead is dominated by the per-call gRPC roundtrip cost, not the payload size.

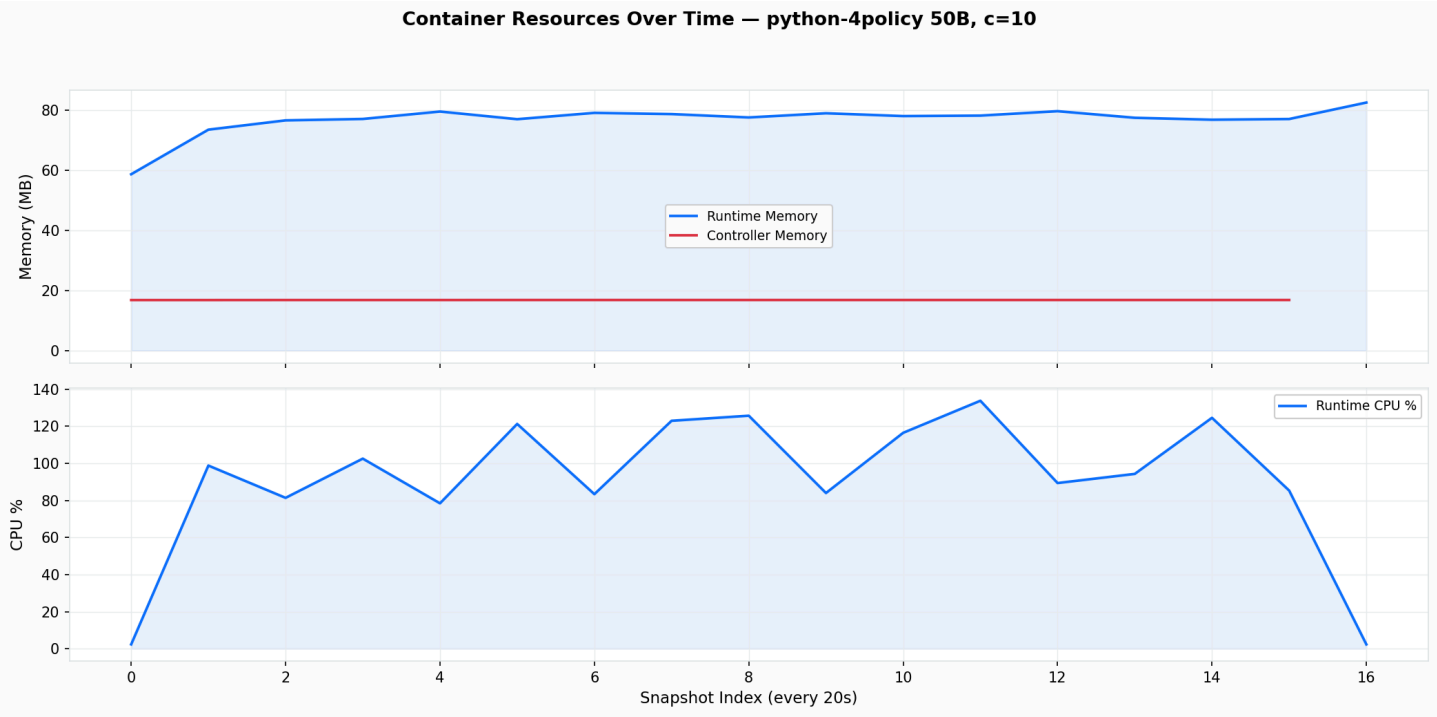
5.7 Resource Usage

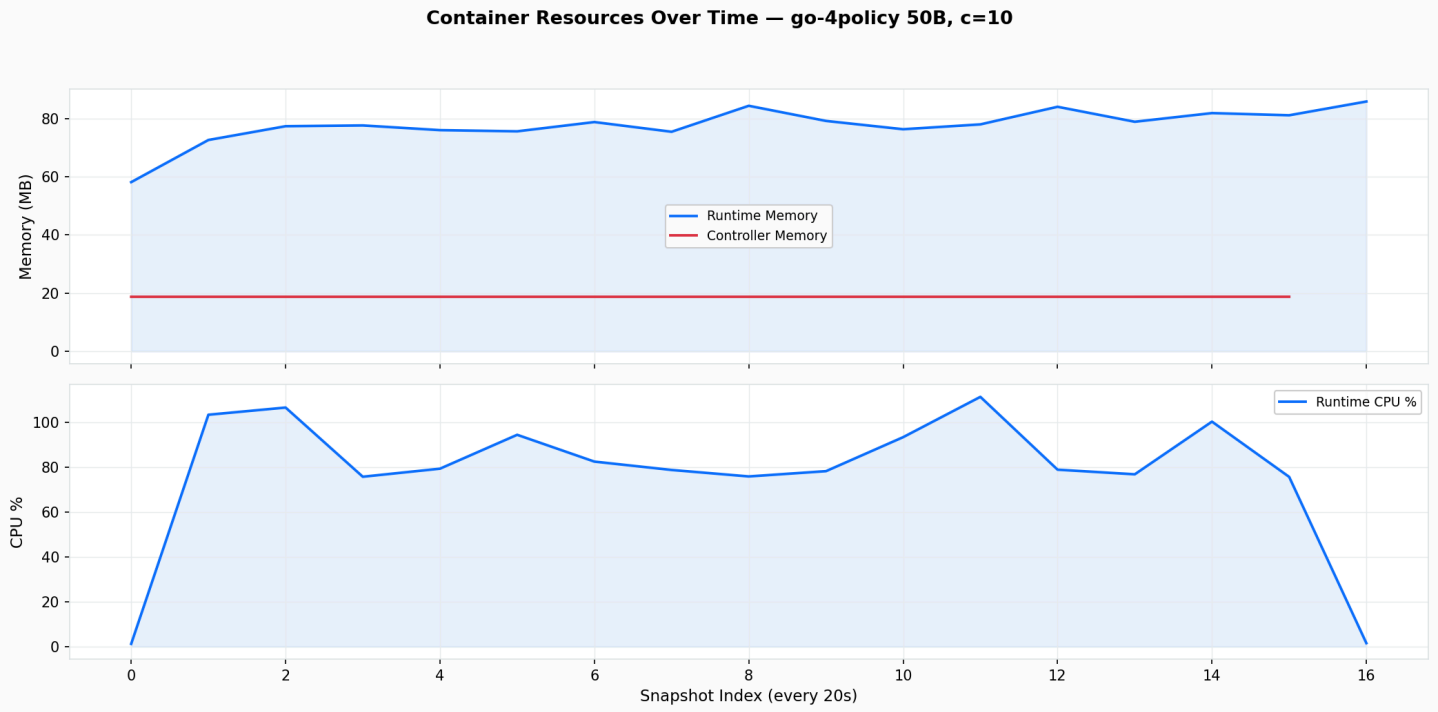


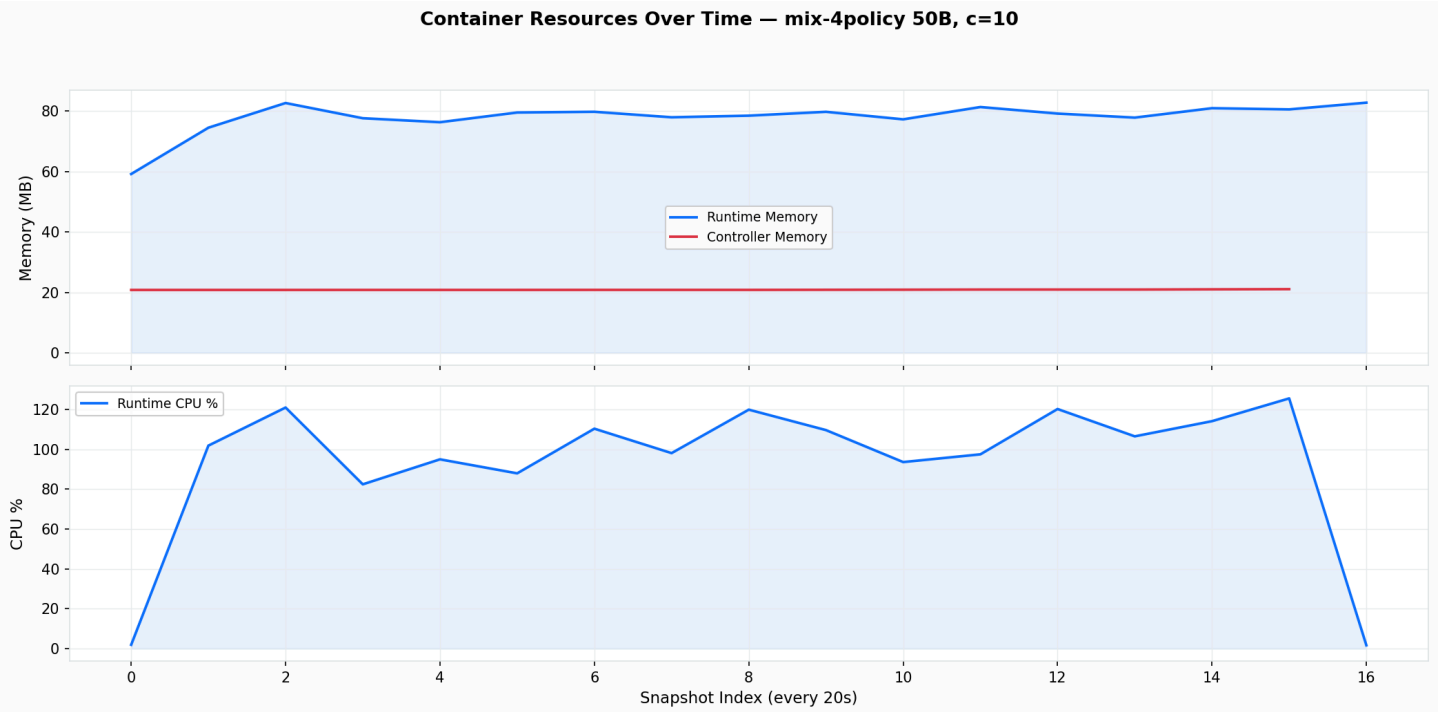
Runtime Container

Concurrency	No-Policies CPU%	Go CPU%	Python CPU%	Mix CPU%	No-Policies Mem (MB)	Go Mem (MB)	Python Mem (MB)	Mix Mem (MB)
1	110.58	111.52	118.11	119.36	77.8	75.5	79.3	73.3
2	116.43	114.42	112.23	132.65	84.1	80.8	75.0	80.4
5	103.19	117.60	134.02	114.95	81.5	86.7	76.7	82.6
10	103.36	111.47	133.77	125.52	86.0	85.9	82.5	82.8

Resource Timelines (Concurrency = 10)





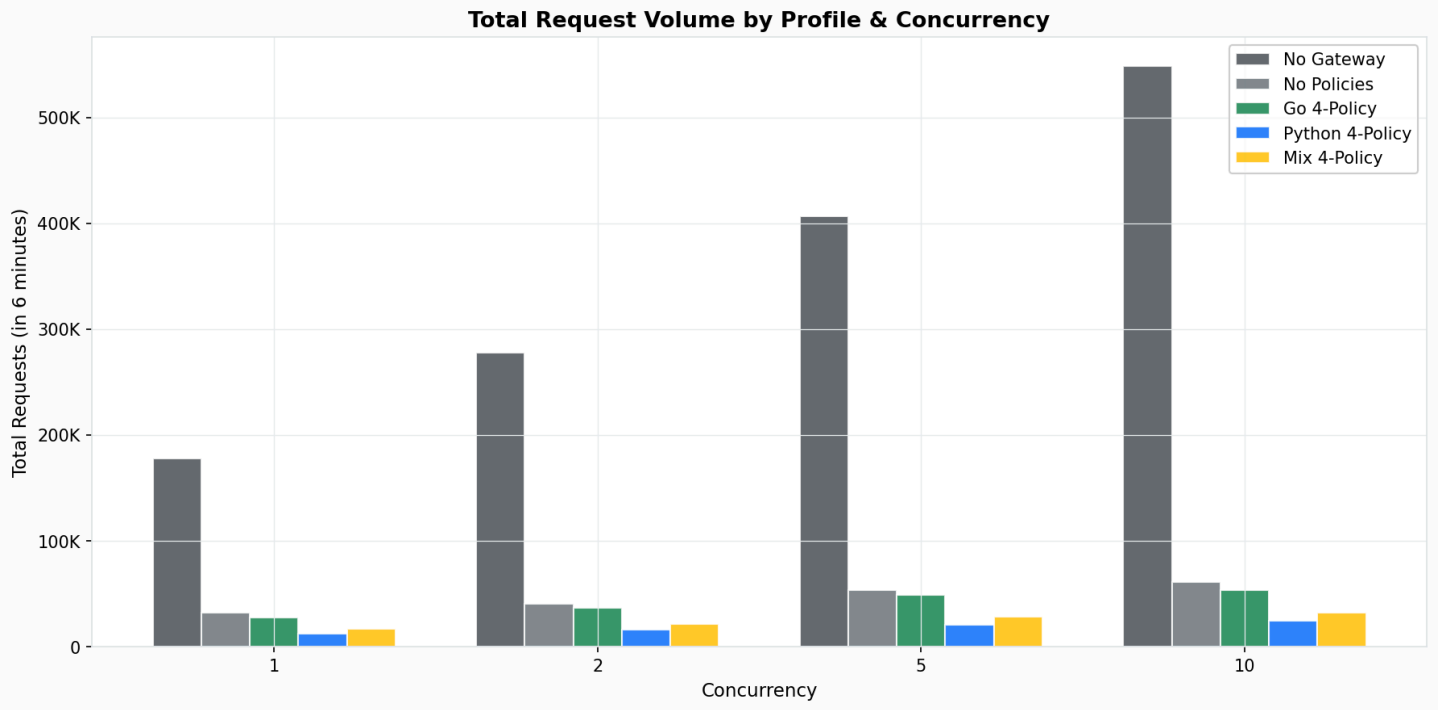


Note

Key Findings:

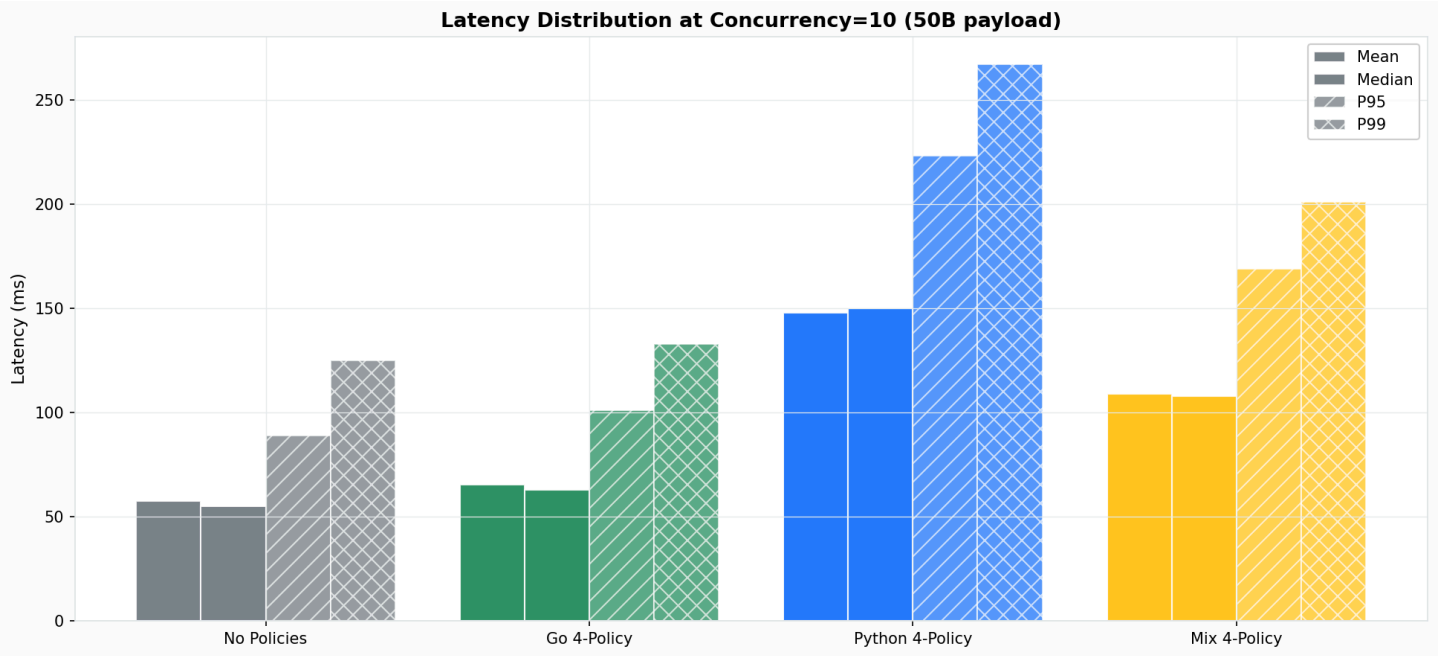
- **Memory is stable** across all profiles and concurrency levels, staying within a tight **73–87 MB** band. No memory leaks detected.
- **CPU peaks** at ~134% for Python at c=10 (using ~67% of the 2 available cores) and ~111% for Go (using ~56%). The system is **not CPU-saturated**.
- The **no-gateway** profile uses negligible CPU (~1.6%), confirming the netty backend imposes virtually zero overhead.
- Resource usage remains **flat and predictable** during the 6-minute test window, with no runaway growth or instability.

5.8 Total Request Volume



Concurrency	No Gateway	No Policies	Go 4-Policy	Python 4-Policy	Mix 4-Policy
1	177,813	32,265	27,705	12,459	16,636
2	277,809	40,728	36,925	16,190	21,521
5	406,598	53,309	48,567	20,748	28,261
10	548,616	61,415	53,335	24,189	32,477

5.9 Latency Distribution at Peak Load (c=10)



6. Accuracy

All 25 tests achieved **100% accuracy** — every request produced the correct policy mutations.

Profile	Payload	Concurrency	Total Requests	Accuracy
no-gateway	50B	1, 2, 5, 10	1,410,836	100.0%
no-gateway	1KiB	1	204,135	100.0%
no-policies	50B	1, 2, 5, 10	187,717	100.0%
no-policies	1KiB	1	29,468	100.0%
go-4policy	50B	1, 2, 5, 10	166,532	100.0%
go-4policy	1KiB	1	27,123	100.0%
python-4policy	50B	1, 2, 5, 10	73,586	100.0%
python-4policy	1KiB	1	12,315	100.0%
mix-4policy	50B	1, 2, 5, 10	98,895	100.0%
mix-4policy	1KiB	1	16,556	100.0%
GRAND TOTAL			**2,227,163**	**100.0%**

> **Tip**
>
> **Over 2.2 million requests** were processed during the benchmark with **zero validation failures**. This confirms the Python executor, Go engine, and mixed chains all produce bit-perfect policy mutations under sustained load.

---

7. Summary of Findings

7.1 Python Executor Performance

Dimension	Result	Verdict
Single-request latency	~17.6ms overhead for a 4-hook chain (vs no-policies)	✅ Acceptable
Per-hook cost	~4.4ms per Python hook (vs ~0.4ms Go)	✅ Acceptable
Overhead ratio	2.6x no-policies baseline — constant across load	✅ Predictable
Payload serialization	<1ms difference between 50B and 1KiB	✅ Excellent
Concurrency scaling	Latency ~2.6x control, stable at all levels	✅ Linear
Throughput at c=10	67.58 TPS (vs Go's 152.55 TPS)	⚠️ 44% of Go

Dimension	Result	Verdict
Memory stability	73–87 MB across all tests	✅ No leaks
CPU utilization	Peak 134% at c=10 (67% of 2 cores)	✅ Headroom
Accuracy	100% across 2.2M requests	✅ Perfect

7.2 Key Takeaways

1. **The Python executor overhead is constant and predictable.** The ~2.6x latency multiplier does not degrade with concurrency — it is a fixed per-call cost dominated by the gRPC bridge roundtrip (~4.4ms per hook).
2. **Payload size has negligible impact.** The 50B → 1KiB transition adds less than 1ms for Python, confirming protobuf serialization is not the bottleneck — the per-call overhead is.
3. **No resource leaks.** Memory stays flat (73–87 MB) across all 25 tests with full gateway restarts between each. CPU utilization is well within limits at ~67% of available cores at peak.
4. **100% correctness under load.** Every single request across 2.2 million total requests produced the expected policy mutations — headers set, bodies prefixed, payloads preserved.
5. **The Go policy engine is nearly free.** At 1.1x the no-policies baseline, Go policies add virtually no overhead — confirming the in-process execution model is extremely efficient.
6. **The Mix profile performs between Go and Python**, as expected. With 2 Go + 2 Python hooks, it achieves ~1.9x the baseline — proportionally weighted by the number of Python hooks.