

zoo-bench performance report

raw data: results/zoo-framework-0.10.0/20261011T093201Z.json

Run environment

item	value
CPU model	AMD EPYC 7763 64-Core Processor
logical cores	4
platform	Linux-x86_64
operating system	Linux 6.17.0-1022-azure (#22-Ubuntu SMP Mon Jul 27 17:24:03 UTC 2026)
Python	CPython 3.13.16
interpreter	/opt/hostedtoolcache/Python/3.13.16/x64/bin/python
GIL	default
framework under test (distribution metadata)	0.10.0
framework under test (module __version__)	0.10.0 (footnote only, not the version criterion)
drive-surface generation of the framework under test	current-generation drive surface (scheduling core + scheduling model; the completion signal comes from the result reactor)
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
command to reproduce	zoo-bench run --framework zoo-framework==0.10.0

dist_version (the distribution metadata) is the source of truth; module_version is only a footnote

Workload

- composition: one unit of work = JSON serialize + deserialize + string concatenation, repeated until the target duration is reached; the tiers are declared by body_tiers_us in matrix.yaml
- **the workload is a stand-in, not a real trace**: its shape resembles business actions such as 'take a frame of data -> serialize -> parse -> transform strings', but this repository holds no real trace, so every conclusion is conditional on that

Measurement methodology

- **the body duration is instrumented within the same run**: the body measures itself internally and reports back; framework overhead = end-to-end - it. **No cross-run subtraction** - two runs differ in state (caches, frequencies, scheduling noise), so subtracting introduces systematic bias.
- **warmups are excluded from statistics**: each unit first runs 3 warmup rounds that are discarded, then only the following 30 measured rounds are counted; the sample count equals 'measured rounds x concurrency'.
- **absolute durations are not comparable across runs**: they only show the order of magnitude; across versions/machines read the **relative quantities within the same run** (overhead share, multiples against the baselines), which are insensitive to overall speed.

Conclusion summary

- Once the framework under test zoo runs a body of about 2700 microseconds or more, its framework overhead drops below 15% of end-to-end; below that tier, the main price of choosing it is the framework overhead itself.

- At concurrency 1 and a 40 microsecond body, the framework under test is 2.11x faster than process_pool (the fastest measured tier at that concurrency).
- At concurrency 1, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 4, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 16, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 64, once the body exceeds about 40 microseconds, apscheduler is no longer faster than the framework under test.
- At concurrency 1, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers asyncio_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, once the body exceeds about 10000 microseconds, bare_thread is no longer faster than the framework under test.
- At concurrency 4, within the measured tiers bare_thread is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, once the body exceeds about 40 microseconds, bare_thread is no longer faster than the framework under test.
- At concurrency 64, within the measured tiers bare_thread is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 4, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 16, within the measured tiers process_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 1, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.

a single-point speedup has no selection meaning; the conclusion is phrased as 'which scheme to choose at what body duration' (the same framework overhead is 70% of a 40 microsecond task and a few percent of a 10

millisecond one)

Overhead threshold crossings

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
apscheduler	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
apscheduler	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
apscheduler	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
apscheduler	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
asyncio_pool	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
asyncio_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
asyncio_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
asyncio_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
bare_thread	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
bare_thread	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
bare_thread	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
bare_thread	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
process_pool	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
process_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
process_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
process_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
thread_pool	1	15.00%	300	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
thread_pool	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
thread_pool	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
thread_pool	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
zoo	1	15.00%	2700	40, 300, 2700, 10000	this is the smallest measured tier that satisfies the condition; the true crossing point lies between it and the previous tier
zoo	4	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
zoo	16	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.
zoo	64	15.00%	-	40, 300, 2700, 10000	this concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values). End-to-end, throughput and the body's self-reported values remain valid, so they are given as usual.

Relative turning points

baseline	concurrency	first tier where it is no longer faster (microseconds)	note
apscheduler	1	40	this is the smallest measured tier that satisfies the condition
apscheduler	4	40	this is the smallest measured tier that satisfies the condition
apscheduler	16	40	this is the smallest measured tier that satisfies the condition
apscheduler	64	40	this is the smallest measured tier that satisfies the condition
asyncio_pool	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	1	10000	this is the smallest measured tier that satisfies the condition
bare_thread	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	16	40	this is the smallest measured tier that satisfies the condition
bare_thread	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
process_pool	1	40	this is the smallest measured tier that satisfies the condition
process_pool	4	40	this is the smallest measured tier that satisfies the condition

baseline	concurrency	first tier where it is no longer faster (microseconds)	note
process_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
process_pool	64	40	this is the smallest measured tier that satisfies the condition
thread_pool	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
thread_pool	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test

Charts

[missing figure: figures/en-overhead_ratio.svg]

Body tiers at concurrency 4 only; ratios hold within a single run. The overhead numbers of concurrency 16, 64 were withheld (beyond this machine's parallelism; see the caliber section) and are not charted

[missing figure: figures/en-throughput.svg]

Body tier 40 microseconds only; throughput holds within a single run

[missing figure: figures/en-relative_multiple.svg]

One panel per measured concurrency (4 in total); multiples hold within a single run

Dimension: Latency percentiles and jitter

end-to-end / concurrency; percentiles and relative spread are given in the same table - a metric that only gives the median is not acceptable

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	157.73 microseconds	288.55 microseconds	140.57 microseconds	141.79 microseconds	332.18 microseconds	83.56 microseconds
1	300	424.91 microseconds	547.57 microseconds	404.67 microseconds	405.40 microseconds	606.50 microseconds	323.16 microseconds
1	2700	2.797 milliseconds	3.120 milliseconds	2.605 milliseconds	2.717 milliseconds	3.017 milliseconds	2.667 milliseconds
1	10000	9.931 milliseconds	9.890 milliseconds	9.476 milliseconds	9.972 milliseconds	10.015 milliseconds	9.618 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
4	40	183.23 microseconds	253.08 microseconds	159.80 microseconds	162.08 microseconds	188.60 microseconds	94.96 microseconds
4	300	439.01 microseconds	519.95 microseconds	410.40 microseconds	409.00 microseconds	295.22 microseconds	362.52 microseconds
4	2700	2.848 milliseconds	3.035 milliseconds	2.749 milliseconds	2.659 milliseconds	1.398 milliseconds	2.706 milliseconds
4	10000	10.050 milliseconds	10.021 milliseconds	9.977 milliseconds	9.793 milliseconds	4.598 milliseconds	9.798 milliseconds
16	40	170.72 microseconds	251.40 microseconds	138.08 microseconds	174.45 microseconds	160.88 microseconds	90.52 microseconds
16	300	439.04 microseconds	501.69 microseconds	394.14 microseconds	430.66 microseconds	280.12 microseconds	351.18 microseconds
16	2700	2.926 milliseconds	2.952 milliseconds	2.822 milliseconds	2.724 milliseconds	1.395 milliseconds	2.718 milliseconds
16	10000	10.507 milliseconds	10.271 milliseconds	9.992 milliseconds	9.834 milliseconds	4.711 milliseconds	10.001 milliseconds
64	40	184.98 microseconds	224.83 microseconds	123.94 microseconds	169.50 microseconds	195.74 microseconds	89.55 microseconds
64	300	472.05 microseconds	505.68 microseconds	373.95 microseconds	424.17 microseconds	297.44 microseconds	349.61 microseconds
64	2700	2.937 milliseconds	2.974 milliseconds	2.777 milliseconds	2.722 milliseconds	1.417 milliseconds	2.768 milliseconds
64	10000	10.515 milliseconds	10.311 milliseconds	10.449 milliseconds	9.812 milliseconds	4.709 milliseconds	10.022 milliseconds

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Framework overhead share

overhead = end-to-end / concurrency - the body duration measured within the same run (no cross-run subtraction). **cells with -**: that concurrency exceeds this machine's parallel capacity; the body's self-reported duration contains scheduling waits from oversubscription and is not comparable with the row's end-to-end per task, so no overhead figure is given (end-to-end and throughput live in their own dimensions)

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	67.93%	82.14%	66.96%	63.81%	82.88%	38.45%
1	300	28.57%	44.63%	26.17%	28.08%	50.03%	10.75%
1	2700	5.17%	9.74%	4.50%	4.20%	11.29%	1.80%
1	10000	1.81%	4.13%	1.41%	1.26%	3.64%	0.60%
4	40	72.88%	77.73%	66.65%	68.11%	49.63%	47.40%
4	300	31.79%	40.54%	25.54%	27.64%	-	15.92%
4	2700	5.27%	9.67%	4.68%	4.32%	-	3.15%
4	10000	-	-	-	-	-	-
16	40	-	-	-	-	-	-
16	300	-	-	-	-	-	-
16	2700	-	-	-	-	-	-
16	10000	-	-	-	-	-	-

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
64	40	-	-	-	-	-	-
64	300	-	-	-	-	-	-
64	2700	-	-	-	-	-	-
64	10000	-	-	-	-	-	-

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Throughput and concurrency scaling

throughput = concurrency / median end-to-end; whether it keeps rising with concurrency is the scaling behaviour

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	6340.1	3465.7	7114.0	7052.9	3010.4	11967.2
1	300	2353.5	1826.2	2471.2	2466.7	1648.8	3094.4
1	2700	357.6	320.5	383.9	368.0	331.4	374.9
1	10000	100.7	101.1	105.5	100.3	99.9	104.0
4	40	21830.7	15805.1	25031.0	24678.8	21208.8	42122.3
4	300	9111.4	7693.1	9746.5	9780.0	13549.0	11034.0
4	2700	1404.6	1317.9	1455.1	1504.0	2861.8	1478.5
4	10000	398.0	399.2	400.9	408.4	869.9	408.3
16	40	93723.0	63644.3	115870.7	91714.4	99452.7	176759.3
16	300	36443.4	31892.4	40594.7	37152.4	57119.0	45560.6
16	2700	5468.7	5419.3	5670.2	5873.4	11467.3	5885.6
16	10000	1522.8	1557.7	1601.3	1626.9	3396.2	1599.9
64	40	345987.8	284656.4	516381.4	377578.2	326969.3	714659.3
64	300	135579.6	126561.3	171144.5	150881.9	215170.2	183061.8
64	2700	21794.5	21519.8	23045.6	23513.0	45160.7	23119.9
64	10000	6086.6	6207.0	6125.1	6522.6	13591.7	6385.7

per-unit values (percentiles, spread, raw per-round samples) are in 'Appendix: all values' at the end.

Dimension: Overhead attribution

scope: splits the per-task end-to-end into submit side / handoff / body / return and, at the same tier and concurrency, **compares each segment against every baseline**, answering 'which segment the excess falls on'. It is **diagnostic, not the main evidence**: it samples a small set of tiers, and does not compare total overhead against the baselines. **A negative handoff is not an error**: it means the submit call had not returned yet while the body was already running - schemes that count thread creation as submit side behave this way (measured: bare_thread handoff -343 microseconds). **The return segment includes waiting for the completion signal** (the last stretch before draining returns); it shares the main measurement's scope

status: ok

this dimension is a **diagnosis**, not the main evidence: the sampled tiers and concurrencies are a fixed small batch, not the whole matrix. The concurrency ceiling is the machine's parallel capacity - beyond it, what is measured is contention;

sampled tiers (microseconds): 40, 10000; concurrency: [1, 4]; this machine's parallel capacity: 4

scheme	concurrency	tier (microseconds)	submit side	handoff	body	return	status
apscheduler	1	40	110.01 microseconds	83.43 microseconds	58.01 microseconds	43.81 microseconds	ok
apscheduler	4	40	101.53 microseconds	508.15 microseconds	56.80 microseconds	221.34 microseconds	ok
apscheduler	1	10000	168.54 microseconds	113.17 microseconds	9.639 milliseconds	63.68 microseconds	ok
apscheduler	4	10000	132.04 microseconds	10.592 milliseconds	14.575 milliseconds	9.825 milliseconds	ok
asyncio_pool	1	40	17.04 microseconds	26.12 microseconds	49.00 microseconds	58.29 microseconds	ok
asyncio_pool	4	40	19.30 microseconds	249.31 microseconds	55.20 microseconds	304.58 microseconds	ok
asyncio_pool	1	10000	26.82 microseconds	41.45 microseconds	9.214 milliseconds	93.47 microseconds	ok
asyncio_pool	4	10000	23.10 microseconds	12.397 milliseconds	9.982 milliseconds	10.142 milliseconds	ok
bare_thread	1	40	146.55 microseconds	-77.58 microseconds	52.32 microseconds	32.58 microseconds	ok
bare_thread	4	40	155.98 microseconds	-75.15 microseconds	51.04 microseconds	317.86 microseconds	ok
bare_thread	1	10000	5.184 milliseconds	-5093.18 microseconds	9.592 milliseconds	41.48 microseconds	ok
bare_thread	4	10000	5.212 milliseconds	-5085.74 microseconds	14.658 milliseconds	9.873 milliseconds	ok
process_pool	1	40	-	-	-	-	not_measurable: the body's start/end timestamps are unavailable - it did not run in this process (cross-process units cannot be split this way)

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
process_pool	4	40	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
process_pool	1	10000	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
process_pool	4	10000	-	-	-	-	not_measura ble: the body's start/end timestamps are unavailable - it did not run in this process (cros s-process units cannot be split this way)
thread_pool	1	40	8.16 microseconds	10.40 microseconds	43.71 microseconds	14.95 microseconds	ok
thread_pool	4	40	11.37 microseconds	174.81 microseconds	47.72 microseconds	141.19 microseconds	ok
thread_pool	1	10000	13.01 microseconds	17.56 microseconds	9.336 milliseconds	28.64 microseconds	ok
thread_pool	4	10000	13.47 microseconds	10.148 milliseconds	15.055 milliseconds	5.076 milliseconds	ok
zoo	1	40	52.47 microseconds	24.98 microseconds	52.22 microseconds	59.61 microseconds	ok
zoo	4	40	62.36 microseconds	303.76 microseconds	46.66 microseconds	275.39 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	92.37 microseconds	35.80 microseconds	9.700 milliseconds	117.86 microseconds	ok
zoo	4	10000	97.22 microseconds	9.968 milliseconds	14.989 milliseconds	9.553 milliseconds	ok

where the excess over each baseline falls

baseline	concurrency	tier (microseconds)	excess per task	mostly on	gap in that segment
apscheduler	1	40	-114.49 microseconds	return	15.80 microseconds
asyncio_pool	1	40	39.74 microseconds	submit side	35.44 microseconds
bare_thread	1	40	34.64 microseconds	handoff	102.56 microseconds
thread_pool	1	40	113.52 microseconds	return	44.66 microseconds
apscheduler	4	40	-235.45 microseconds	return	54.05 microseconds
asyncio_pool	4	40	23.70 microseconds	handoff	54.46 microseconds
bare_thread	4	40	223.28 microseconds	handoff	378.91 microseconds
thread_pool	4	40	304.11 microseconds	return	134.20 microseconds
apscheduler	1	10000	-54.37 microseconds	body	61.78 microseconds
asyncio_pool	1	10000	565.33 microseconds	body	486.14 microseconds
bare_thread	1	10000	229.49 microseconds	handoff	5.129 milliseconds
thread_pool	1	10000	541.75 microseconds	body	364.01 microseconds
apscheduler	4	10000	-424.87 microseconds	body	414.07 microseconds
asyncio_pool	4	10000	-419.34 microseconds	body	5.007 milliseconds
bare_thread	4	10000	8.528 milliseconds	handoff	15.053 milliseconds
thread_pool	4	10000	1.09 microseconds	return	4.477 milliseconds

instrumentation seams used (temporarily wrapped on the harness side, no framework code changed):

BaseWaiter._dispatch_worker (handing a worker to the scheduling model);

WorkerDispatchCore.resolve_period / resolve_phase / resolve_run_timeout (per-round policy lookups);

BaseWaiter.execute_service (one scheduling round)

the cost of instrumentation itself (on the framework-under-test side, paired from two rounds in the same subprocess, so it can be discounted honestly): zoo 40 microseconds / concurrency 1: without instrumentation 186.64 microseconds vs with 191.03 microseconds, a difference of 4.39 microseconds; zoo 40 microseconds / concurrency 4: without instrumentation 671.61 microseconds vs with 673.41 microseconds, a difference of

1.79 microseconds; zoo 10000 microseconds / concurrency 1: without instrumentation 9.915 milliseconds vs with 9.953 milliseconds, a difference of 37.32 microseconds; zoo 10000 microseconds / concurrency 4: without instrumentation 39.693 milliseconds vs with 39.568 milliseconds, a difference of 125.02 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost)

additivity of the sub-measurements: the sub-measurements are **independent measurements**: each one's duration contains its own preemption waits, so their sum can exceed the submit side they all live in (about 3% measured). Do not read them as an additive partition. measured overlap this round: zoo 10000 microseconds / concurrency 1 overlaps 11.8%; zoo 10000 microseconds / concurrency 4 overlaps 93.5%

what the submit side of the framework under test consists of

- concurrency 1, tier 40 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.05 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 22.43 microseconds; per-round policy lookups (period / phase / timeout each read config) 20.74 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 8.43 microseconds
- concurrency 4, tier 40 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.00 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 31.12 microseconds; per-round policy lookups (period / phase / timeout each read config) 28.63 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.32 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 1.26 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 37.35 microseconds; per-round policy lookups (period / phase / timeout each read config) 37.74 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 15.28 microseconds
- concurrency 4, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.00 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 43.77 microseconds; per-round policy lookups (period / phase / timeout each read config) 54.88 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 5.48 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-114 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 40 microseconds more per task than `asyncio_pool`, most of it on 'submit side' (+35 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -1 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 35 microseconds more per task than `bare_thread`, most of it on 'handoff' (+103 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -94 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 114 microseconds more per task than `thread_pool`, most of it on 'return' (+45 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 0, dispatch (handed to the scheduling model, incl. pool enqueue) 22, per-round policy lookups (period / phase / timeout each read config) 21, submit side, other (the adapter's own bookkeeping and completion-signal state) 8.

- At concurrency 4 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-235 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 24 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+54 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'return' is -29 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 223 microseconds more per task than `bare_thread`, most of it on 'handoff' (+379 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -94 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 304 microseconds more per task than `thread_pool`, most of it on 'return' (+134 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -1 microseconds, so read them together.
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 0, dispatch (handed to the scheduling model, incl. pool enqueue) 31, per-round policy lookups (period / phase / timeout each read config) 29, submit side, other (the adapter's own bookkeeping and completion-signal state) 4.
- At concurrency 1 and a 10000 microsecond body, the framework under test is not slower overall than `apscheduler` (-54 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 565 microseconds more per task than `asyncio_pool`, most of it on 'body' (+486 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -6 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 229 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5129 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5092 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 542 microseconds more per task than `thread_pool`, most of it on 'body' (+364 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 1, dispatch (handed to the scheduling model, incl. pool enqueue) 37, per-round policy lookups (period / phase / timeout each read config) 38, submit side, other (the adapter's own bookkeeping and completion-signal state) 15.
- At concurrency 4 and a 10000 microsecond body, the framework under test is not slower overall than `apscheduler` (-425 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 4 and a 10000 microsecond body, the framework under test is not slower overall than `asyncio_pool` (-419 microseconds per task): the two sides have different timing structures, and the per-segment differences have both signs, so no 'which segment' is given.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 8528 microseconds more per task than `bare_thread`, most of it on 'handoff' (+15053 microseconds).

- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5114 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 1 microseconds more per task than thread_pool, most of it on 'return' (+4477 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -180 microseconds, so read them together.
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 0, dispatch (handed to the scheduling model, incl. pool enqueue) 44, per-round policy lookups (period / phase / timeout each read config) 55, submit side, other (the adapter's own bookkeeping and completion-signal state) 5.

Dimension: The cost of scheduling semantics

scope: this dimension uses an **internal switch comparison** (same workload, semantics all off vs. enabled one by one) and **does not compare across schemes** - a bare writeup has no counterpart for priority / timeout / retry, so it can only answer 'what is this framework's semantics worth', not 'how much more expensive than a bare writeup'

status: ok (the conclusion was reached on the 'current' drive-surface generation installed during this measurement)

semantics item	layer	how probed	measurable	evidence / reason
timeout	dispatch layer	behavioral probe (driven repeatedly through scheduling rounds; measures the moment the worker leaves the in-flight table)	yes	timeout reap observed: the worker left the in-flight table before the body ended (0.0123s) (body 0.15s, run_timeout 0.01s); judgment action at dispatch_core.py:427; what is in effect is observation and circuit-breaking (recording, marking unhealthy, removing the in-flight registration, stopping dispatch), not termination of the still-running worker - CPython cannot safely interrupt a running thread

semantics item	layer	how probed	measurable	evidence / reason
priority	event layer	structural probe (checks whether the symbols required for execution exist)	no	not re-checked: this item's conclusion was reached on the 'previous' generation; taking it as this generation's fact asserts wiring in the report that the current generation does not have. Re-deriving it needs a probe design of its own (each item needs clean on/off states before it can be priced)
retry	event layer	structural probe (checks whether the symbols required for execution exist)	no	not re-checked: this item's conclusion was reached on the 'previous' generation; taking it as this generation's fact asserts wiring in the report that the current generation does not have. Re-deriving it needs a probe design of its own (each item needs clean on/off states before it can be priced)

this conclusion was reached on the version under test and its coordinates hold for that version only; after appending a version to matrix.yaml, re-run this probe

Dimension: Native task line: three execution forms of the same subject

scope: this line is measured on its own: it enters neither the main matrix's pivot tables nor its crossings, and its numbers must not be read against the baseline adapters listed there

verdict: not_applicable

the framework under test has no such capability (symbols not found: adaptive_params, dual_arm_worker, native_params, native_task_worker) - this line does not apply to this version, and that is not a failure

equivalence gate: not run (the probe did not reach it)

loaded task modbus_rtu.parse_response - Modbus RTU response-frame parsing: the input is a bare RTU ADU response frame (no transport framing), and the tier's frame bytes are its length

tiers R1, R8, R125 (frame bytes 7, 21, 255); concurrencies 1, 4, 16, 64; each cell runs 100 warm-up plus 100 measured rounds, one child process per concurrency (4 in all)

tiers where the framework under test is faster

Tiers where the framework under test is ahead (the two sides differ by more than that tier's band). This comes from the **same within-run comparison** as the openly published unfavourable data - one side takes the faster tiers, the other the slower ones, so both cover the same ground and neither can pick its own tiers. **This**

section being empty is not a failure: it should be empty when nothing is distinguishable or when the framework loses everywhere.

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
apscheduler	1	40	the framework under test is 1.83x faster than apscheduler	157.73 microseconds
process_pool	1	40	the framework under test is 2.11x faster than process_pool	157.73 microseconds
apscheduler	1	300	the framework under test is 1.29x faster than apscheduler	424.91 microseconds
process_pool	1	300	the framework under test is 1.43x faster than process_pool	424.91 microseconds
apscheduler	1	2700	the framework under test is 1.12x faster than apscheduler	2.797 milliseconds
process_pool	1	2700	the framework under test is 1.08x faster than process_pool	2.797 milliseconds
apscheduler	4	40	the framework under test is 1.38x faster than apscheduler	183.23 microseconds
apscheduler	4	300	the framework under test is 1.18x faster than apscheduler	439.01 microseconds
apscheduler	4	2700	the framework under test is 1.07x faster than apscheduler	2.848 milliseconds
apscheduler	16	40	the framework under test is 1.47x faster than apscheduler	170.72 microseconds
apscheduler	16	300	the framework under test is 1.14x faster than apscheduler	439.04 microseconds
apscheduler	64	40	the framework under test is 1.22x faster than apscheduler	184.98 microseconds

Openly published unfavourable data

Tiers where the framework under test is behind (the two sides differ by more than that tier's band). **A report missing this part is not acceptable** - a report that only shows where it wins gives the reader every reason to distrust it. Tiers whose difference is smaller than the band are not here but in the 'no winner' section: that is the verdict's outcome, not a place to hide things.

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	1	40	thread_pool is 1.89x faster than zoo	157.73 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	1	300	thread_pool is 1.31x faster than zoo	424.91 microseconds
asyncio_pool	1	2700	asyncio_pool is 1.07x faster than zoo	2.797 milliseconds
bare_thread	1	2700	bare_thread is 1.03x faster than zoo	2.797 milliseconds
thread_pool	1	2700	thread_pool is 1.05x faster than zoo	2.797 milliseconds
thread_pool	1	10000	thread_pool is 1.03x faster than zoo	9.931 milliseconds
thread_pool	4	40	thread_pool is 1.93x faster than zoo	183.23 microseconds
asyncio_pool	4	300	asyncio_pool is 1.07x faster than zoo	439.01 microseconds
bare_thread	4	300	bare_thread is 1.07x faster than zoo	439.01 microseconds
process_pool	4	300	process_pool is 1.49x faster than zoo	439.01 microseconds
thread_pool	4	300	thread_pool is 1.21x faster than zoo	439.01 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.04x faster than zoo	2.848 milliseconds
bare_thread	4	2700	bare_thread is 1.07x faster than zoo	2.848 milliseconds
process_pool	4	2700	process_pool is 2.04x faster than zoo	2.848 milliseconds
thread_pool	4	2700	thread_pool is 1.05x faster than zoo	2.848 milliseconds
process_pool	4	10000	process_pool is 2.19x faster than zoo	10.050 milliseconds
thread_pool	4	10000	thread_pool is 1.03x faster than zoo	10.050 milliseconds
thread_pool	16	40	thread_pool is 1.89x faster than zoo	170.72 microseconds
asyncio_pool	16	300	asyncio_pool is 1.11x faster than zoo	439.04 microseconds
process_pool	16	300	process_pool is 1.57x faster than zoo	439.04 microseconds
thread_pool	16	300	thread_pool is 1.25x faster than zoo	439.04 microseconds
asyncio_pool	16	2700	asyncio_pool is 1.04x faster than zoo	2.926 milliseconds
bare_thread	16	2700	bare_thread is 1.07x faster than zoo	2.926 milliseconds
process_pool	16	2700	process_pool is 2.10x faster than zoo	2.926 milliseconds
thread_pool	16	2700	thread_pool is 1.08x faster than zoo	2.926 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
asyncio_pool	16	10000	asyncio_pool is 1.05x faster than zoo	10.507 milliseconds
bare_thread	16	10000	bare_thread is 1.07x faster than zoo	10.507 milliseconds
process_pool	16	10000	process_pool is 2.23x faster than zoo	10.507 milliseconds
thread_pool	16	10000	thread_pool is 1.05x faster than zoo	10.507 milliseconds
thread_pool	64	40	thread_pool is 2.07x faster than zoo	184.98 microseconds
asyncio_pool	64	300	asyncio_pool is 1.26x faster than zoo	472.05 microseconds
bare_thread	64	300	bare_thread is 1.11x faster than zoo	472.05 microseconds
process_pool	64	300	process_pool is 1.59x faster than zoo	472.05 microseconds
thread_pool	64	300	thread_pool is 1.35x faster than zoo	472.05 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.06x faster than zoo	2.937 milliseconds
bare_thread	64	2700	bare_thread is 1.08x faster than zoo	2.937 milliseconds
process_pool	64	2700	process_pool is 2.07x faster than zoo	2.937 milliseconds
thread_pool	64	2700	thread_pool is 1.06x faster than zoo	2.937 milliseconds
bare_thread	64	10000	bare_thread is 1.07x faster than zoo	10.515 milliseconds
process_pool	64	10000	process_pool is 2.23x faster than zoo	10.515 milliseconds
thread_pool	64	10000	thread_pool is 1.05x faster than zoo	10.515 milliseconds

tiers with no winner

Tiers where the two sides **cannot be told apart**: the difference is smaller than that tier's band. The band comes from the two sides' own across-round spread (stdev / median of the per-task medians) **within the same run**, so it varies with the sample and the reader can recompute it from the archived per-round samples. It also covers every measured tier and states a reason for each - **this section is not a place to hide things**: reporting noise-level differences as '1.00x faster' is.

criterion: band = the framework's own across-round relative spread plus the baseline's; a single side's across-round relative spread = stdev / median of that unit's per-round per-task medians

band this run: 1.0% to 33.3% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	1	40	0.89x	the two sides differ by 10.9%, less than the band of 19.3% - no winner
bare_thread	1	40	0.90x	the two sides differ by 10.1%, less than the band of 16.8% - no winner
asyncio_pool	1	300	0.95x	the two sides differ by 4.8%, less than the band of 9.9% - no winner
bare_thread	1	300	0.95x	the two sides differ by 4.6%, less than the band of 9.8% - no winner
apscheduler	1	10000	1.00x	the two sides differ by 0.4%, less than the band of 1.3% - no winner
asyncio_pool	1	10000	0.95x	the two sides differ by 4.6%, less than the band of 6.3% - no winner
bare_thread	1	10000	1.00x	the two sides differ by 0.4%, less than the band of 1.0% - no winner
process_pool	1	10000	1.01x	the two sides differ by 0.8%, less than the band of 1.3% - no winner
asyncio_pool	4	40	0.87x	the two sides differ by 12.8%, less than the band of 17.9% - no winner
bare_thread	4	40	0.88x	the two sides differ by 11.5%, less than the band of 13.1% - no winner
process_pool	4	40	1.03x	the two sides differ by 2.9%, less than the band of 26.2% - no winner
apscheduler	4	10000	1.00x	the two sides differ by 0.3%, less than the band of 2.1% - no winner
asyncio_pool	4	10000	0.99x	the two sides differ by 0.7%, less than the band of 1.6% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
bare_thread	4	10000	0.97x	the two sides differ by 2.6%, less than the band of 3.9% - no winner
asyncio_pool	16	40	0.81x	the two sides differ by 19.1%, less than the band of 22.7% - no winner
bare_thread	16	40	1.02x	the two sides differ by 2.2%, less than the band of 18.1% - no winner
process_pool	16	40	0.94x	the two sides differ by 5.8%, less than the band of 33.0% - no winner
bare_thread	16	300	0.98x	the two sides differ by 1.9%, less than the band of 6.6% - no winner
apscheduler	16	2700	1.01x	the two sides differ by 0.9%, less than the band of 2.0% - no winner
apscheduler	16	10000	0.98x	the two sides differ by 2.2%, less than the band of 4.4% - no winner
asyncio_pool	64	40	0.67x	the two sides differ by 33.0%, less than the band of 33.3% - no winner
bare_thread	64	40	0.92x	the two sides differ by 8.4%, less than the band of 12.7% - no winner
process_pool	64	40	1.06x	the two sides differ by 5.8%, less than the band of 22.3% - no winner
apscheduler	64	300	1.07x	the two sides differ by 7.1%, less than the band of 8.4% - no winner
apscheduler	64	2700	1.01x	the two sides differ by 1.3%, less than the band of 4.2% - no winner
apscheduler	64	10000	0.98x	the two sides differ by 1.9%, less than the band of 5.1% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	64	10000	0.99x	the two sides differ by 0.6%, less than the band of 4.5% - no winner

caliber limitations and sources of bias

- **measured layer:** dispatch layer: drives BaseWaiter.execute_service() directly (thread-pool mode) instead of Master's full run loop - the latter's service loop fires once per second, which would lift the latency scale from microseconds to seconds. **One scheduling round per submitted task**, so that whole round's bookkeeping is charged to this single task; Master's usage serves several workers per round, which would spread that cost over tasks
- **scope note** (apscheduler): APScheduler's scheduled-job model, including job registration and trigger-decision cost; the library swallows exceptions raised inside jobs, so this tier re-raises them during drain. **Caliber note:** the library offers no job-completion notification, so the completion signal comes from this tier's Condition, and the cost of one notification per task counts toward its end-to-end
- **scope note** (asyncio_pool): the asyncio event loop dispatches **synchronous** bodies (equivalent to loop.run_in_executor), not native coroutine concurrency; the difference from a thread pool is the event loop's own cost
- **scope note** (bare_thread): one threading.Thread per task, no pooling and no reuse; thread creation cost counts toward end-to-end. Bodies' self-reported timings are collected with a lock-free list.append (atomic under the GIL) - **instrumentation must not charge one tier a cost other tiers do not pay**, or the comparison becomes a comparison of instrumentation
- **scope note** (process_pool): ProcessPoolExecutor: pickling the body and its return value across processes counts toward end-to-end, so this tier is not directly comparable to the multi-threaded tiers
- **scope note** (thread_pool): concurrent.futures.ThreadPoolExecutor, pool size = concurrency; threads are reused, with no task-queue semantics
- **scope note** (zoo): end-to-end stops at the worker completing and being deregistered from the in-flight table; the framework's later result delivery splits by generation: the previous generation's delivery path is broken (dispatch looks reactors up by reactor_name rather than by topic, so every call raises), and the completion signal can only come from the framework's internal callback; the current generation has that fixed, and the completion signal comes from the framework's own result reactor. **Caliber note:** the completion signal comes from this tier's Condition, and the cost of one notification per task counts toward its end-to-end; a bias in that direction makes the framework under test look slower, i.e. it works against it rather than for it
- **groups with no overhead figure:** for the (adapter, concurrency) groups below, concurrency exceeds the parallel capacity of the machine this run was on: the body's self-reported duration contains scheduling waits from oversubscription, which is not the same thing as 'wall clock / concurrency', so subtracting them is meaningless (measured, it produced negative values and meaningless change factors). This report therefore **gives them no framework overhead and no crossing point**; end-to-end, throughput and the body's self-reported values remain valid and not one field has been dropped from the raw archive. The criterion is the logical core count from the environment self-report, plus one evidential backstop: a computed negative overhead is likewise treated as unreadable - a framework can only add time, never remove it
- **native task line: no verdict:** the native task line came out not_applicable (the framework under test has no such capability (symbols not found: adaptive_params, dual_arm_worker, native_params, native_task_worker) - this line does not apply to this version, and that is not a failure) and produced no figures. It is reported here

rather than folded into the self-check: one line's failure must not hold back the other versions' data or the main matrix

- **self-check items excluded from the verdict:** for these units the body-tier deviation did not take part in the verdict (concurrency above the lowest, or a tier so short that wall clock is dominated by scheduling granularity). The raw values are still in place, but the wall clock at those two places is not enough to conclude whether the calibration is right - **factor this in when reading those tiers' numbers**

absolute durations: valid only within a single run; do not compare across runs

bring your own baseline

the baselines of this harness are **open**: implement the four methods of `zoo_bench.adapters.BaseAdapter` (setup / submit / drain / teardown), register with `zoo_bench.adapters.registry.register`, or list your module path in `extra_modules` of `matrix.yaml` - **no file in this repository needs to change**. Your implementation goes through the same equivalence verification (submit N bodies, assert all ran and each exactly once); failing it means the data is not trusted. **We deliberately write this path into the report**: a report written by the framework's own maintainer, measuring that framework, gives every reason to suspect a slowed baseline or a chosen caliber; letting anyone submit their own baseline is what makes this report challengeable and falsifiable.

self-check

- overall measurement self-check: passed
- process isolation: {'parent_pid': 56883, 'child_pids': [57338, 57341, 57344, 57347, 57350, 57356, 57362, 57368, 57375, 57393, 57411, 57429, 57447, 57505, 57563, 57620, 57688, 57690, 57692, 57694, 57696, 57701, 57706, 57711, 57716, 57733, 57750, 57768, 57789, 57849, 57908, 57973, 58040, 58074, 58108, 58142, 58176, 58309, 58442, 58575, 58708, 59237, 59766, 60296, 60825, 62938, 65051, 67164, 69280, 69284, 69288, 69292, 69296, 69303, 69310, 69317, 69324, 69343, 69362, 69381, 69400, 69467, 69535, 69603, 69670, 69672, 69674, 69676, 69678, 69683, 69688, 69693, 69698, 69715, 69732, 69749, 69766, 69822, 69879, 69940, 70005, 70007, 70009, 70011, 70013, 70018, 70023, 70028, 70033, 70050, 70067, 70084, 70102, 70167, 70232, 70297], 'all_distinct': True, 'none_equals_parent': True}

Appendix: all values

the body carries only pivot tables for readability; this appendix gives **every field as measured before any trimming** per unit (percentiles, spread, measured body durations, ...). **Not one value is dropped**: the details live inside the report body itself, not behind links, folds or attachments - 'verifiable' that requires a separate lookup is not verifiable

Appendix: Latency percentiles and jitter

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	1	40	288.55 microseconds	329.21 microseconds	348.86 microseconds	0.070
apscheduler	1	300	547.57 microseconds	582.23 microseconds	591.17 microseconds	0.031
apscheduler	1	2700	3.120 milliseconds	3.300 milliseconds	3.323 milliseconds	0.028
apscheduler	1	10000	9.890 milliseconds	10.004 milliseconds	10.068 milliseconds	0.007

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	4	40	253.08 microseconds	265.53 microseconds	317.11 microseconds	0.063
apscheduler	4	300	519.95 microseconds	547.62 microseconds	560.45 microseconds	0.024
apscheduler	4	2700	3.035 milliseconds	3.090 milliseconds	3.115 milliseconds	0.013
apscheduler	4	10000	10.021 milliseconds	10.167 milliseconds	10.479 milliseconds	0.013
apscheduler	16	40	251.40 microseconds	266.50 microseconds	270.78 microseconds	0.033
apscheduler	16	300	501.69 microseconds	532.97 microseconds	561.46 microseconds	0.034
apscheduler	16	2700	2.952 milliseconds	3.018 milliseconds	3.039 milliseconds	0.011
apscheduler	16	10000	10.271 milliseconds	10.573 milliseconds	11.250 milliseconds	0.024
apscheduler	64	40	224.83 microseconds	252.58 microseconds	268.72 microseconds	0.064
apscheduler	64	300	505.68 microseconds	591.84 microseconds	596.80 microseconds	0.064
apscheduler	64	2700	2.974 milliseconds	3.060 milliseconds	3.248 milliseconds	0.023
apscheduler	64	10000	10.311 milliseconds	10.844 milliseconds	10.870 milliseconds	0.027
asyncio_pool	1	40	140.57 microseconds	171.45 microseconds	177.08 microseconds	0.092
asyncio_pool	1	300	404.67 microseconds	456.89 microseconds	485.11 microseconds	0.061
asyncio_pool	1	2700	2.605 milliseconds	2.662 milliseconds	2.662 milliseconds	0.013
asyncio_pool	1	10000	9.476 milliseconds	9.716 milliseconds	11.633 milliseconds	0.057
asyncio_pool	4	40	159.80 microseconds	183.66 microseconds	186.21 microseconds	0.098
asyncio_pool	4	300	410.40 microseconds	426.74 microseconds	442.25 microseconds	0.026
asyncio_pool	4	2700	2.749 milliseconds	2.791 milliseconds	2.832 milliseconds	0.010
asyncio_pool	4	10000	9.977 milliseconds	10.058 milliseconds	10.063 milliseconds	0.009
asyncio_pool	16	40	138.08 microseconds	155.96 microseconds	163.23 microseconds	0.073
asyncio_pool	16	300	394.14 microseconds	402.93 microseconds	421.98 microseconds	0.021
asyncio_pool	16	2700	2.822 milliseconds	2.867 milliseconds	2.934 milliseconds	0.012
asyncio_pool	16	10000	9.992 milliseconds	10.044 milliseconds	10.143 milliseconds	0.005

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
asyncio_pool	64	40	123.94 microseconds	210.21 microseconds	217.01 microseconds	0.225
asyncio_pool	64	300	373.95 microseconds	381.84 microseconds	408.52 microseconds	0.024
asyncio_pool	64	2700	2.777 milliseconds	2.813 milliseconds	2.815 milliseconds	0.008
asyncio_pool	64	10000	10.449 milliseconds	10.665 milliseconds	10.843 milliseconds	0.021
bare_thread	1	40	141.79 microseconds	164.71 microseconds	170.32 microseconds	0.067
bare_thread	1	300	405.40 microseconds	449.66 microseconds	458.71 microseconds	0.060
bare_thread	1	2700	2.717 milliseconds	2.750 milliseconds	2.785 milliseconds	0.009
bare_thread	1	10000	9.972 milliseconds	10.057 milliseconds	10.074 milliseconds	0.004
bare_thread	4	40	162.08 microseconds	176.75 microseconds	185.28 microseconds	0.049
bare_thread	4	300	409.00 microseconds	423.72 microseconds	429.34 microseconds	0.017
bare_thread	4	2700	2.659 milliseconds	2.692 milliseconds	2.693 milliseconds	0.005
bare_thread	4	10000	9.793 milliseconds	10.400 milliseconds	11.034 milliseconds	0.031
bare_thread	16	40	174.45 microseconds	182.12 microseconds	188.43 microseconds	0.027
bare_thread	16	300	430.66 microseconds	440.99 microseconds	442.80 microseconds	0.014
bare_thread	16	2700	2.724 milliseconds	2.770 milliseconds	2.774 milliseconds	0.008
bare_thread	16	10000	9.834 milliseconds	9.899 milliseconds	10.052 milliseconds	0.006
bare_thread	64	40	169.50 microseconds	176.72 microseconds	178.09 microseconds	0.019
bare_thread	64	300	424.17 microseconds	430.10 microseconds	432.47 microseconds	0.009
bare_thread	64	2700	2.722 milliseconds	2.741 milliseconds	2.744 milliseconds	0.005
bare_thread	64	10000	9.812 milliseconds	9.871 milliseconds	9.945 milliseconds	0.004
process_pool	1	40	332.18 microseconds	372.02 microseconds	385.43 microseconds	0.057
process_pool	1	300	606.50 microseconds	644.08 microseconds	651.86 microseconds	0.035
process_pool	1	2700	3.017 milliseconds	3.063 milliseconds	3.187 milliseconds	0.014
process_pool	1	10000	10.015 milliseconds	10.125 milliseconds	10.215 milliseconds	0.007

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
process_pool	4	40	188.60 microseconds	247.46 microseconds	253.31 microseconds	0.180
process_pool	4	300	295.22 microseconds	373.91 microseconds	401.46 microseconds	0.112
process_pool	4	2700	1.398 milliseconds	1.700 milliseconds	2.011 milliseconds	0.107
process_pool	4	10000	4.598 milliseconds	5.151 milliseconds	5.619 milliseconds	0.056
process_pool	16	40	160.88 microseconds	220.91 microseconds	241.56 microseconds	0.175
process_pool	16	300	280.12 microseconds	321.58 microseconds	336.26 microseconds	0.083
process_pool	16	2700	1.395 milliseconds	1.464 milliseconds	1.476 milliseconds	0.026
process_pool	16	10000	4.711 milliseconds	4.815 milliseconds	4.940 milliseconds	0.015
process_pool	64	40	195.74 microseconds	242.18 microseconds	253.66 microseconds	0.115
process_pool	64	300	297.44 microseconds	327.42 microseconds	331.38 microseconds	0.056
process_pool	64	2700	1.417 milliseconds	1.538 milliseconds	1.615 milliseconds	0.039
process_pool	64	10000	4.709 milliseconds	4.749 milliseconds	4.799 milliseconds	0.005
thread_pool	1	40	83.56 microseconds	99.22 microseconds	104.74 microseconds	0.104
thread_pool	1	300	323.16 microseconds	358.34 microseconds	360.98 microseconds	0.048
thread_pool	1	2700	2.667 milliseconds	2.692 milliseconds	2.741 milliseconds	0.008
thread_pool	1	10000	9.618 milliseconds	9.832 milliseconds	10.008 milliseconds	0.011
thread_pool	4	40	94.96 microseconds	107.50 microseconds	111.37 microseconds	0.072
thread_pool	4	300	362.52 microseconds	370.91 microseconds	372.59 microseconds	0.019
thread_pool	4	2700	2.706 milliseconds	2.753 milliseconds	2.776 milliseconds	0.013
thread_pool	4	10000	9.798 milliseconds	9.901 milliseconds	9.981 milliseconds	0.008
thread_pool	16	40	90.52 microseconds	104.07 microseconds	112.32 microseconds	0.070
thread_pool	16	300	351.18 microseconds	363.24 microseconds	369.43 microseconds	0.020
thread_pool	16	2700	2.718 milliseconds	2.758 milliseconds	2.904 milliseconds	0.019
thread_pool	16	10000	10.001 milliseconds	10.427 milliseconds	10.716 milliseconds	0.019

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
thread_pool	64	40	89.55 microseconds	91.29 microseconds	91.83 microseconds	0.017
thread_pool	64	300	349.61 microseconds	355.29 microseconds	356.04 microseconds	0.008
thread_pool	64	2700	2.768 milliseconds	2.814 milliseconds	2.828 milliseconds	0.007
thread_pool	64	10000	10.022 milliseconds	10.175 milliseconds	10.246 milliseconds	0.014
zoo	1	40	157.73 microseconds	190.47 microseconds	212.68 microseconds	0.101
zoo	1	300	424.91 microseconds	455.46 microseconds	458.23 microseconds	0.038
zoo	1	2700	2.797 milliseconds	2.822 milliseconds	2.825 milliseconds	0.006
zoo	1	10000	9.931 milliseconds	10.029 milliseconds	10.091 milliseconds	0.006
zoo	4	40	183.23 microseconds	208.27 microseconds	212.35 microseconds	0.081
zoo	4	300	439.01 microseconds	459.84 microseconds	497.54 microseconds	0.038
zoo	4	2700	2.848 milliseconds	2.913 milliseconds	2.938 milliseconds	0.012
zoo	4	10000	10.050 milliseconds	10.172 milliseconds	10.227 milliseconds	0.008
zoo	16	40	170.72 microseconds	225.48 microseconds	232.07 microseconds	0.154
zoo	16	300	439.04 microseconds	483.56 microseconds	497.31 microseconds	0.052
zoo	16	2700	2.926 milliseconds	2.968 milliseconds	2.985 milliseconds	0.010
zoo	16	10000	10.507 milliseconds	10.987 milliseconds	11.238 milliseconds	0.020
zoo	64	40	184.98 microseconds	222.63 microseconds	225.59 microseconds	0.108
zoo	64	300	472.05 microseconds	487.34 microseconds	496.03 microseconds	0.020
zoo	64	2700	2.937 milliseconds	3.067 milliseconds	3.105 milliseconds	0.019
zoo	64	10000	10.515 milliseconds	11.025 milliseconds	11.049 milliseconds	0.024

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	237.02 microseconds	82.14%	51.52 microseconds
apscheduler	1	300	244.40 microseconds	44.63%	303.17 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	2700	304.02 microseconds	9.74%	2.816 milliseconds
apscheduler	1	10000	408.51 microseconds	4.13%	9.482 milliseconds
apscheduler	4	40	196.71 microseconds	77.73%	56.37 microseconds
apscheduler	4	300	210.79 microseconds	40.54%	309.16 microseconds
apscheduler	4	2700	293.59 microseconds	9.67%	2.741 milliseconds
apscheduler	4	10000	-	-	14.823 milliseconds
apscheduler	16	40	-	-	53.43 microseconds
apscheduler	16	300	-	-	309.91 microseconds
apscheduler	16	2700	-	-	2.713 milliseconds
apscheduler	16	10000	-	-	20.657 milliseconds
apscheduler	64	40	-	-	50.89 microseconds
apscheduler	64	300	-	-	297.16 microseconds
apscheduler	64	2700	-	-	2.718 milliseconds
apscheduler	64	10000	-	-	20.896 milliseconds
asyncio_pool	1	40	94.13 microseconds	66.96%	46.44 microseconds
asyncio_pool	1	300	105.90 microseconds	26.17%	298.76 microseconds
asyncio_pool	1	2700	117.31 microseconds	4.50%	2.487 milliseconds
asyncio_pool	1	10000	133.41 microseconds	1.41%	9.342 milliseconds
asyncio_pool	4	40	106.51 microseconds	66.65%	53.29 microseconds
asyncio_pool	4	300	104.83 microseconds	25.54%	305.58 microseconds
asyncio_pool	4	2700	128.74 microseconds	4.68%	2.620 milliseconds
asyncio_pool	4	10000	-	-	14.654 milliseconds
asyncio_pool	16	40	-	-	53.84 microseconds
asyncio_pool	16	300	-	-	303.55 microseconds
asyncio_pool	16	2700	-	-	2.698 milliseconds
asyncio_pool	16	10000	-	-	24.790 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
asyncio_pool	64	40	-	-	52.47 microseconds
asyncio_pool	64	300	-	-	296.86 microseconds
asyncio_pool	64	2700	-	-	2.643 milliseconds
asyncio_pool	64	10000	-	-	86.213 milliseconds
bare_thread	1	40	90.48 microseconds	63.81%	51.31 microseconds
bare_thread	1	300	113.83 microseconds	28.08%	291.57 microseconds
bare_thread	1	2700	114.16 microseconds	4.20%	2.603 milliseconds
bare_thread	1	10000	125.91 microseconds	1.26%	9.846 milliseconds
bare_thread	4	40	110.40 microseconds	68.11%	51.69 microseconds
bare_thread	4	300	113.04 microseconds	27.64%	295.95 microseconds
bare_thread	4	2700	114.79 microseconds	4.32%	2.545 milliseconds
bare_thread	4	10000	-	-	14.784 milliseconds
bare_thread	16	40	-	-	50.76 microseconds
bare_thread	16	300	-	-	299.23 microseconds
bare_thread	16	2700	-	-	2.594 milliseconds
bare_thread	16	10000	-	-	14.902 milliseconds
bare_thread	64	40	-	-	50.72 microseconds
bare_thread	64	300	-	-	296.82 microseconds
bare_thread	64	2700	-	-	2.584 milliseconds
bare_thread	64	10000	-	-	14.905 milliseconds
process_pool	1	40	275.31 microseconds	82.88%	56.87 microseconds
process_pool	1	300	303.41 microseconds	50.03%	303.09 microseconds
process_pool	1	2700	340.55 microseconds	11.29%	2.677 milliseconds
process_pool	1	10000	364.82 microseconds	3.64%	9.650 milliseconds
process_pool	4	40	93.61 microseconds	49.63%	94.99 microseconds
process_pool	4	300	-	-	569.90 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
process_pool	4	2700	-	-	4.831 milliseconds
process_pool	4	10000	-	-	17.613 milliseconds
process_pool	16	40	-	-	90.15 microseconds
process_pool	16	300	-	-	564.29 microseconds
process_pool	16	2700	-	-	9.624 milliseconds
process_pool	16	10000	-	-	53.594 milliseconds
process_pool	64	40	-	-	95.71 microseconds
process_pool	64	300	-	-	572.83 microseconds
process_pool	64	2700	-	-	25.932 milliseconds
process_pool	64	10000	-	-	173.835 milliseconds
thread_pool	1	40	32.13 microseconds	38.45%	51.44 microseconds
thread_pool	1	300	34.74 microseconds	10.75%	288.43 microseconds
thread_pool	1	2700	48.01 microseconds	1.80%	2.619 milliseconds
thread_pool	1	10000	57.70 microseconds	0.60%	9.560 milliseconds
thread_pool	4	40	45.01 microseconds	47.40%	49.95 microseconds
thread_pool	4	300	57.72 microseconds	15.92%	304.79 microseconds
thread_pool	4	2700	85.16 microseconds	3.15%	2.620 milliseconds
thread_pool	4	10000	-	-	19.499 milliseconds
thread_pool	16	40	-	-	48.21 microseconds
thread_pool	16	300	-	-	297.14 microseconds
thread_pool	16	2700	-	-	2.632 milliseconds
thread_pool	16	10000	-	-	20.219 milliseconds
thread_pool	64	40	-	-	47.21 microseconds
thread_pool	64	300	-	-	293.01 microseconds
thread_pool	64	2700	-	-	2.681 milliseconds
thread_pool	64	10000	-	-	25.119 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
zoo	1	40	107.14 microseconds	67.93%	50.59 microseconds
zoo	1	300	121.39 microseconds	28.57%	303.52 microseconds
zoo	1	2700	144.58 microseconds	5.17%	2.652 milliseconds
zoo	1	10000	179.89 microseconds	1.81%	9.751 milliseconds
zoo	4	40	133.53 microseconds	72.88%	49.70 microseconds
zoo	4	300	139.58 microseconds	31.79%	299.43 microseconds
zoo	4	2700	150.03 microseconds	5.27%	2.698 milliseconds
zoo	4	10000	-	-	15.037 milliseconds
zoo	16	40	-	-	44.68 microseconds
zoo	16	300	-	-	301.40 microseconds
zoo	16	2700	-	-	2.756 milliseconds
zoo	16	10000	-	-	51.225 milliseconds
zoo	64	40	-	-	47.40 microseconds
zoo	64	300	-	-	301.28 microseconds
zoo	64	2700	-	-	2.753 milliseconds
zoo	64	10000	-	-	82.491 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3465.7
apscheduler	1	300	1826.2
apscheduler	1	2700	320.5
apscheduler	1	10000	101.1
apscheduler	4	40	15805.1
apscheduler	4	300	7693.1
apscheduler	4	2700	1317.9
apscheduler	4	10000	399.2
apscheduler	16	40	63644.3
apscheduler	16	300	31892.4
apscheduler	16	2700	5419.3
apscheduler	16	10000	1557.7
apscheduler	64	40	284656.4

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	64	300	126561.3
apscheduler	64	2700	21519.8
apscheduler	64	10000	6207.0
asyncio_pool	1	40	7114.0
asyncio_pool	1	300	2471.2
asyncio_pool	1	2700	383.9
asyncio_pool	1	10000	105.5
asyncio_pool	4	40	25031.0
asyncio_pool	4	300	9746.5
asyncio_pool	4	2700	1455.1
asyncio_pool	4	10000	400.9
asyncio_pool	16	40	115870.7
asyncio_pool	16	300	40594.7
asyncio_pool	16	2700	5670.2
asyncio_pool	16	10000	1601.3
asyncio_pool	64	40	516381.4
asyncio_pool	64	300	171144.5
asyncio_pool	64	2700	23045.6
asyncio_pool	64	10000	6125.1
bare_thread	1	40	7052.9
bare_thread	1	300	2466.7
bare_thread	1	2700	368.0
bare_thread	1	10000	100.3
bare_thread	4	40	24678.8
bare_thread	4	300	9780.0
bare_thread	4	2700	1504.0
bare_thread	4	10000	408.4
bare_thread	16	40	91714.4
bare_thread	16	300	37152.4
bare_thread	16	2700	5873.4
bare_thread	16	10000	1626.9
bare_thread	64	40	377578.2
bare_thread	64	300	150881.9
bare_thread	64	2700	23513.0
bare_thread	64	10000	6522.6
process_pool	1	40	3010.4
process_pool	1	300	1648.8
process_pool	1	2700	331.4
process_pool	1	10000	99.9
process_pool	4	40	21208.8
process_pool	4	300	13549.0
process_pool	4	2700	2861.8
process_pool	4	10000	869.9
process_pool	16	40	99452.7

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
process_pool	16	300	57119.0
process_pool	16	2700	11467.3
process_pool	16	10000	3396.2
process_pool	64	40	326969.3
process_pool	64	300	215170.2
process_pool	64	2700	45160.7
process_pool	64	10000	13591.7
thread_pool	1	40	11967.2
thread_pool	1	300	3094.4
thread_pool	1	2700	374.9
thread_pool	1	10000	104.0
thread_pool	4	40	42122.3
thread_pool	4	300	11034.0
thread_pool	4	2700	1478.5
thread_pool	4	10000	408.3
thread_pool	16	40	176759.3
thread_pool	16	300	45560.6
thread_pool	16	2700	5885.6
thread_pool	16	10000	1599.9
thread_pool	64	40	714659.3
thread_pool	64	300	183061.8
thread_pool	64	2700	23119.9
thread_pool	64	10000	6385.7
zoo	1	40	6340.1
zoo	1	300	2353.5
zoo	1	2700	357.6
zoo	1	10000	100.7
zoo	4	40	21830.7
zoo	4	300	9111.4
zoo	4	2700	1404.6
zoo	4	10000	398.0
zoo	16	40	93723.0
zoo	16	300	36443.4
zoo	16	2700	5468.7
zoo	16	10000	1522.8
zoo	64	40	345987.8
zoo	64	300	135579.6
zoo	64	2700	21794.5
zoo	64	10000	6086.6