

zoo-bench performance report

raw data: results/zoo-framework-0.9.0/20261011T092749Z.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)	0.9.0
framework under test (module __version__)	0.9.0 (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==0.9.0

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 1.83x 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, 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 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, 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 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, 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 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, 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 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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
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	-	not observed: within the measured tiers this baseline is always faster than the framework under test

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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
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	182.57 microseconds	288.52 microseconds	140.37 microseconds	148.91 microseconds	333.53 microseconds	75.77 microseconds
1	300	449.17 microseconds	542.76 microseconds	386.47 microseconds	384.68 microseconds	607.37 microseconds	322.77 microseconds
1	2700	2.754 milliseconds	2.999 milliseconds	2.752 milliseconds	2.672 milliseconds	3.009 milliseconds	2.689 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	10000	10.066 milliseconds	9.971 milliseconds	9.482 milliseconds	9.648 milliseconds	9.888 milliseconds	9.361 milliseconds
4	40	206.14 microseconds	256.36 microseconds	173.04 microseconds	199.87 microseconds	168.78 microseconds	96.03 microseconds
4	300	485.26 microseconds	511.06 microseconds	407.48 microseconds	419.42 microseconds	304.29 microseconds	350.77 microseconds
4	2700	2.946 milliseconds	2.983 milliseconds	2.755 milliseconds	2.706 milliseconds	1.385 milliseconds	2.742 milliseconds
4	10000	10.106 milliseconds	10.258 milliseconds	9.867 milliseconds	9.696 milliseconds	4.621 milliseconds	9.813 milliseconds
16	40	205.75 microseconds	224.01 microseconds	134.08 microseconds	167.78 microseconds	149.92 microseconds	90.21 microseconds
16	300	470.57 microseconds	505.14 microseconds	393.00 microseconds	424.88 microseconds	278.51 microseconds	343.15 microseconds
16	2700	2.899 milliseconds	2.980 milliseconds	2.802 milliseconds	2.711 milliseconds	1.370 milliseconds	2.772 milliseconds
16	10000	10.084 milliseconds	9.881 milliseconds	9.935 milliseconds	9.609 milliseconds	4.741 milliseconds	9.803 milliseconds
64	40	207.28 microseconds	220.85 microseconds	120.26 microseconds	171.52 microseconds	194.08 microseconds	89.21 microseconds
64	300	491.01 microseconds	501.75 microseconds	384.40 microseconds	424.30 microseconds	299.76 microseconds	350.35 microseconds
64	2700	2.953 milliseconds	2.965 milliseconds	2.811 milliseconds	2.675 milliseconds	1.364 milliseconds	2.696 milliseconds
64	10000	10.538 milliseconds	10.167 milliseconds	10.407 milliseconds	9.827 milliseconds	4.686 milliseconds	10.244 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	71.66%	79.81%	67.08%	64.13%	83.82%	41.38%
1	300	32.12%	44.93%	26.24%	25.34%	48.87%	11.10%
1	2700	5.65%	9.02%	4.58%	4.02%	11.37%	2.35%
1	10000	2.55%	3.19%	1.54%	1.25%	3.45%	0.67%
4	40	73.26%	78.08%	68.70%	70.52%	67.43%	48.25%
4	300	36.28%	39.64%	25.85%	28.21%	-	14.92%
4	2700	7.68%	8.71%	4.45%	4.05%	-	2.85%
4	10000	-	-	-	-	-	-
16	40	-	-	-	-	-	-
16	300	-	-	-	-	-	-
16	2700	-	-	-	-	-	-

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
16	10000	-	-	-	-	-	-
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	5477.4	3466.0	7123.9	6715.5	2998.2	13198.4
1	300	2226.3	1842.4	2587.5	2599.6	1646.4	3098.1
1	2700	363.1	333.4	363.4	374.3	332.3	371.9
1	10000	99.3	100.3	105.5	103.6	101.1	106.8
4	40	19404.4	15602.9	23116.5	20013.0	23699.2	41654.8
4	300	8242.9	7826.8	9816.5	9537.0	13145.4	11403.4
4	2700	1357.7	1341.0	1451.9	1478.1	2888.8	1459.0
4	10000	395.8	389.9	405.4	412.6	865.5	407.6
16	40	77763.4	71424.4	119331.0	95364.7	106727.0	177361.2
16	300	34001.3	31674.5	40712.6	37658.1	57448.9	46626.5
16	2700	5518.7	5369.8	5711.1	5901.5	11678.3	5772.6
16	10000	1586.7	1619.2	1610.5	1665.2	3375.1	1632.1
64	40	308754.3	289794.7	532160.1	373124.9	329761.1	717389.6
64	300	130343.7	127553.4	166491.4	150836.5	213501.9	182674.1
64	2700	21671.4	21585.8	22767.3	23921.9	46909.7	23741.9
64	10000	6073.5	6294.8	6149.5	6512.8	13657.9	6247.3

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	108.78 microseconds	81.29 microseconds	58.86 microseconds	42.86 microseconds	ok
apscheduler	4	40	110.68 microseconds	519.28 microseconds	56.18 microseconds	218.97 microseconds	ok
apscheduler	1	10000	166.72 microseconds	122.62 microseconds	9.782 milliseconds	59.54 microseconds	ok
apscheduler	4	10000	119.74 microseconds	12.884 milliseconds	14.542 milliseconds	9.805 milliseconds	ok
asyncio_pool	1	40	27.40 microseconds	47.40 microseconds	56.28 microseconds	78.36 microseconds	ok
asyncio_pool	4	40	19.00 microseconds	259.15 microseconds	54.13 microseconds	309.16 microseconds	ok
asyncio_pool	1	10000	22.65 microseconds	36.13 microseconds	9.197 milliseconds	84.50 microseconds	ok
asyncio_pool	4	10000	23.82 microseconds	12.044 milliseconds	14.259 milliseconds	9.649 milliseconds	ok
bare_thread	1	40	141.09 microseconds	-77.35 microseconds	52.12 microseconds	31.62 microseconds	ok
bare_thread	4	40	150.71 microseconds	-74.46 microseconds	50.79 microseconds	302.37 microseconds	ok
bare_thread	1	10000	5.173 milliseconds	-5094.45 microseconds	9.675 milliseconds	38.67 microseconds	ok
bare_thread	4	10000	5.189 milliseconds	-5084.41 microseconds	14.786 milliseconds	9.975 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	8.04 microseconds	10.46 microseconds	45.22 microseconds	15.50 microseconds	ok
thread_pool	4	40	11.13 microseconds	180.20 microseconds	49.39 microseconds	148.07 microseconds	ok
thread_pool	1	10000	13.34 microseconds	17.54 microseconds	9.434 milliseconds	28.94 microseconds	ok
thread_pool	4	10000	13.42 microseconds	10.048 milliseconds	15.032 milliseconds	9.237 milliseconds	ok
zoo	1	40	56.37 microseconds	24.72 microseconds	52.56 microseconds	62.97 microseconds	ok
zoo	4	40	64.44 microseconds	392.36 microseconds	56.29 microseconds	334.47 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	93.87 microseconds	30.59 microseconds	9.703 milliseconds	100.45 microseconds	ok
zoo	4	10000	92.78 microseconds	10.328 milliseconds	14.664 milliseconds	9.974 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	-100.74 microseconds	return	20.10 microseconds
asyncio_pool	1	40	-13.24 microseconds	submit side	28.96 microseconds
bare_thread	1	40	49.06 microseconds	handoff	102.07 microseconds
thread_pool	1	40	117.53 microseconds	submit side	48.33 microseconds
apscheduler	4	40	-134.18 microseconds	return	115.50 microseconds
asyncio_pool	4	40	134.12 microseconds	handoff	133.22 microseconds
bare_thread	4	40	337.46 microseconds	handoff	466.82 microseconds
thread_pool	4	40	404.51 microseconds	handoff	212.16 microseconds
apscheduler	1	10000	-196.87 microseconds	return	40.91 microseconds
asyncio_pool	1	10000	580.65 microseconds	body	506.69 microseconds
bare_thread	1	10000	141.08 microseconds	handoff	5.125 milliseconds
thread_pool	1	10000	429.60 microseconds	body	269.02 microseconds
apscheduler	4	10000	257.50 microseconds	return	169.34 microseconds
asyncio_pool	4	10000	1.275 milliseconds	body	404.28 microseconds
bare_thread	4	10000	8.348 milliseconds	handoff	15.412 milliseconds
thread_pool	4	10000	671.61 microseconds	return	737.02 microseconds

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 195.82 microseconds vs with 197.14 microseconds, a difference of 1.32 microseconds; zoo 40 microseconds /

concurrency 4: without instrumentation 788.53 microseconds vs with 782.07 microseconds, a difference of 6.46 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 1: without instrumentation 9.971 milliseconds vs with 9.936 milliseconds, a difference of 34.78 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 4: without instrumentation 40.412 milliseconds vs with 40.009 milliseconds, a difference of 403.79 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 40 microseconds / concurrency 1 overlaps 1.3%; zoo 40 microseconds / concurrency 4 overlaps 5.1%; zoo 10000 microseconds / concurrency 4 overlaps 94.1%

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) 0.25 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 26.68 microseconds; per-round policy lookups (period / phase / timeout each read config) 20.73 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 8.48 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) 36.68 microseconds; per-round policy lookups (period / phase / timeout each read config) 29.29 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.30 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 0.30 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 43.21 microseconds; per-round policy lookups (period / phase / timeout each read config) 35.41 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 15.10 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) 51.42 microseconds; per-round policy lookups (period / phase / timeout each read config) 49.10 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 3.68 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-101 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 is not slower overall than `asyncio_pool` (-13 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 49 microseconds more per task than `bare_thread`, most of it on 'handoff' (+102 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -85 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 118 microseconds more per task than `thread_pool`, most of it on 'submit side' (+48 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) 27, 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` (-134 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 134 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+133 microseconds).
- At concurrency 4 and a 40 microsecond body, the framework under test spends 337 microseconds more per task than `bare_thread`, most of it on 'handoff' (+467 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -86 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 405 microseconds more per task than `thread_pool`, most of it on 'handoff' (+212 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) 37, 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` (-197 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 581 microseconds more per task than `asyncio_pool`, most of it on 'body' (+507 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 141 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5125 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5079 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 430 microseconds more per task than `thread_pool`, most of it on 'body' (+269 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) 43, per-round policy lookups (period / phase / timeout each read config) 35, 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 spends 258 microseconds more per task than `apscheduler`, most of it on 'return' (+169 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -2557 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 1275 microseconds more per task than `asyncio_pool`, most of it on 'body' (+404 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -1716 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 8348 microseconds more per task than `bare_thread`, most of it on 'handoff' (+15412 microseconds).

- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5096 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 672 microseconds more per task than thread_pool, most of it on 'return' (+737 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -368 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) 51, per-round policy lookups (period / phase / timeout each read config) 49, submit side, other (the adapter's own bookkeeping and completion-signal state) 4.

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 (0.0122s) (body 0.15s, run_timeout 0.01s); judgment action at dispatch_core.py:425; 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 1.58x faster than apscheduler	182.57 microseconds
process_pool	1	40	the framework under test is 1.83x faster than process_pool	182.57 microseconds
apscheduler	1	300	the framework under test is 1.21x faster than apscheduler	449.17 microseconds
process_pool	1	300	the framework under test is 1.35x faster than process_pool	449.17 microseconds
apscheduler	1	2700	the framework under test is 1.09x faster than apscheduler	2.754 milliseconds
process_pool	1	2700	the framework under test is 1.09x faster than process_pool	2.754 milliseconds
apscheduler	4	40	the framework under test is 1.24x faster than apscheduler	206.14 microseconds
apscheduler	16	300	the framework under test is 1.07x faster than apscheduler	470.57 microseconds
apscheduler	16	2700	the framework under test is 1.03x faster than apscheduler	2.899 milliseconds

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
asyncio_pool	1	40	asyncio_pool is 1.30x faster than zoo	182.57 microseconds
thread_pool	1	40	thread_pool is 2.41x faster than zoo	182.57 microseconds
asyncio_pool	1	300	asyncio_pool is 1.16x faster than zoo	449.17 microseconds
bare_thread	1	300	bare_thread is 1.17x faster than zoo	449.17 microseconds
thread_pool	1	300	thread_pool is 1.39x faster than zoo	449.17 microseconds
bare_thread	1	2700	bare_thread is 1.03x faster than zoo	2.754 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	1	2700	thread_pool is 1.02x faster than zoo	2.754 milliseconds
asyncio_pool	1	10000	asyncio_pool is 1.06x faster than zoo	10.066 milliseconds
bare_thread	1	10000	bare_thread is 1.04x faster than zoo	10.066 milliseconds
thread_pool	1	10000	thread_pool is 1.08x faster than zoo	10.066 milliseconds
thread_pool	4	40	thread_pool is 2.15x faster than zoo	206.14 microseconds
asyncio_pool	4	300	asyncio_pool is 1.19x faster than zoo	485.26 microseconds
bare_thread	4	300	bare_thread is 1.16x faster than zoo	485.26 microseconds
process_pool	4	300	process_pool is 1.59x faster than zoo	485.26 microseconds
thread_pool	4	300	thread_pool is 1.38x faster than zoo	485.26 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.07x faster than zoo	2.946 milliseconds
bare_thread	4	2700	bare_thread is 1.09x faster than zoo	2.946 milliseconds
process_pool	4	2700	process_pool is 2.13x faster than zoo	2.946 milliseconds
thread_pool	4	2700	thread_pool is 1.07x faster than zoo	2.946 milliseconds
asyncio_pool	4	10000	asyncio_pool is 1.02x faster than zoo	10.106 milliseconds
process_pool	4	10000	process_pool is 2.19x faster than zoo	10.106 milliseconds
thread_pool	4	10000	thread_pool is 1.03x faster than zoo	10.106 milliseconds
asyncio_pool	16	40	asyncio_pool is 1.53x faster than zoo	205.75 microseconds
bare_thread	16	40	bare_thread is 1.23x faster than zoo	205.75 microseconds
thread_pool	16	40	thread_pool is 2.28x faster than zoo	205.75 microseconds
asyncio_pool	16	300	asyncio_pool is 1.20x faster than zoo	470.57 microseconds
bare_thread	16	300	bare_thread is 1.11x faster than zoo	470.57 microseconds
process_pool	16	300	process_pool is 1.69x faster than zoo	470.57 microseconds
thread_pool	16	300	thread_pool is 1.37x faster than zoo	470.57 microseconds
asyncio_pool	16	2700	asyncio_pool is 1.03x faster than zoo	2.899 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
bare_thread	16	2700	bare_thread is 1.07x faster than zoo	2.899 milliseconds
process_pool	16	2700	process_pool is 2.12x faster than zoo	2.899 milliseconds
thread_pool	16	2700	thread_pool is 1.05x faster than zoo	2.899 milliseconds
apscheduler	16	10000	apscheduler is 1.02x faster than zoo	10.084 milliseconds
asyncio_pool	16	10000	asyncio_pool is 1.02x faster than zoo	10.084 milliseconds
bare_thread	16	10000	bare_thread is 1.05x faster than zoo	10.084 milliseconds
process_pool	16	10000	process_pool is 2.13x faster than zoo	10.084 milliseconds
thread_pool	16	10000	thread_pool is 1.03x faster than zoo	10.084 milliseconds
asyncio_pool	64	40	asyncio_pool is 1.72x faster than zoo	207.28 microseconds
bare_thread	64	40	bare_thread is 1.21x faster than zoo	207.28 microseconds
thread_pool	64	40	thread_pool is 2.32x faster than zoo	207.28 microseconds
asyncio_pool	64	300	asyncio_pool is 1.28x faster than zoo	491.01 microseconds
bare_thread	64	300	bare_thread is 1.16x faster than zoo	491.01 microseconds
process_pool	64	300	process_pool is 1.64x faster than zoo	491.01 microseconds
thread_pool	64	300	thread_pool is 1.40x faster than zoo	491.01 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.05x faster than zoo	2.953 milliseconds
bare_thread	64	2700	bare_thread is 1.10x faster than zoo	2.953 milliseconds
process_pool	64	2700	process_pool is 2.16x faster than zoo	2.953 milliseconds
thread_pool	64	2700	thread_pool is 1.10x faster than zoo	2.953 milliseconds
apscheduler	64	10000	apscheduler is 1.04x faster than zoo	10.538 milliseconds
bare_thread	64	10000	bare_thread is 1.07x faster than zoo	10.538 milliseconds
process_pool	64	10000	process_pool is 2.25x faster than zoo	10.538 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: 1.4% to 30.1% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
bare_thread	1	40	0.82x	the two sides differ by 18.4%, less than the band of 30.1% - no winner
asyncio_pool	1	2700	1.00x	the two sides differ by 0.1%, less than the band of 3.1% - no winner
apscheduler	1	10000	0.99x	the two sides differ by 0.9%, less than the band of 4.6% - no winner
process_pool	1	10000	0.98x	the two sides differ by 1.8%, less than the band of 1.9% - no winner
asyncio_pool	4	40	0.84x	the two sides differ by 16.1%, less than the band of 17.3% - no winner
bare_thread	4	40	0.97x	the two sides differ by 3.0%, less than the band of 14.3% - no winner
process_pool	4	40	0.82x	the two sides differ by 18.1%, less than the band of 22.2% - no winner
apscheduler	4	300	1.05x	the two sides differ by 5.3%, less than the band of 7.0% - no winner
apscheduler	4	2700	1.01x	the two sides differ by 1.3%, less than the band of 2.7% - no winner
apscheduler	4	10000	1.02x	the two sides differ by 1.5%, less than the band of 3.2% - no winner
bare_thread	4	10000	0.96x	the two sides differ by 4.1%, less than the band of 4.7% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
apscheduler	16	40	1.09x	the two sides differ by 8.9%, less than the band of 15.6% - no winner
process_pool	16	40	0.73x	the two sides differ by 27.1%, less than the band of 29.0% - no winner
apscheduler	64	40	1.07x	the two sides differ by 6.5%, less than the band of 9.9% - no winner
process_pool	64	40	0.94x	the two sides differ by 6.4%, less than the band of 16.1% - no winner
apscheduler	64	300	1.02x	the two sides differ by 2.2%, less than the band of 5.0% - no winner
apscheduler	64	2700	1.00x	the two sides differ by 0.4%, less than the band of 1.7% - no winner
asyncio_pool	64	10000	0.99x	the two sides differ by 1.2%, less than the band of 3.9% - no winner
thread_pool	64	10000	0.97x	the two sides differ by 2.8%, less than the band of 3.7% - 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': 43405, 'child_pids': [43869, 43872, 43875, 43878, 43881, 43887, 43893, 43899, 43905, 43923, 43941, 43959, 43978, 44034, 44090, 44145, 44206, 44208, 44210, 44212, 44214, 44219, 44224, 44229, 44234, 44251, 44268, 44285, 44302, 44363, 44425, 44490, 44555, 44589, 44623,

44657, 44691, 44824, 44959, 45092, 45225, 45754, 46283, 46812, 47341, 49454, 51567, 53680, 55794, 55798, 55802, 55806, 55810, 55817, 55824, 55831, 55838, 55857, 55876, 55895, 55914, 55981, 56048, 56115, 56182, 56184, 56186, 56189, 56191, 56196, 56201, 56206, 56211, 56228, 56245, 56262, 56279, 56335, 56392, 56453, 56519, 56521, 56523, 56525, 56527, 56532, 56537, 56542, 56547, 56564, 56581, 56598, 56615, 56664, 56713, 56758], '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	288.52 microseconds	319.90 microseconds	323.17 microseconds	0.053
apscheduler	1	300	542.76 microseconds	584.20 microseconds	596.60 microseconds	0.042
apscheduler	1	2700	2.999 milliseconds	3.199 milliseconds	3.209 milliseconds	0.027
apscheduler	1	10000	9.971 milliseconds	10.397 milliseconds	11.362 milliseconds	0.034
apscheduler	4	40	256.36 microseconds	278.20 microseconds	299.54 microseconds	0.047
apscheduler	4	300	511.06 microseconds	526.87 microseconds	571.99 microseconds	0.030
apscheduler	4	2700	2.983 milliseconds	3.051 milliseconds	3.097 milliseconds	0.012
apscheduler	4	10000	10.258 milliseconds	10.480 milliseconds	11.282 milliseconds	0.025
apscheduler	16	40	224.01 microseconds	265.51 microseconds	278.59 microseconds	0.084
apscheduler	16	300	505.14 microseconds	532.88 microseconds	551.13 microseconds	0.026
apscheduler	16	2700	2.980 milliseconds	3.025 milliseconds	3.042 milliseconds	0.010
apscheduler	16	10000	9.881 milliseconds	9.964 milliseconds	9.988 milliseconds	0.008
apscheduler	64	40	220.85 microseconds	246.89 microseconds	281.47 microseconds	0.068
apscheduler	64	300	501.75 microseconds	548.43 microseconds	550.58 microseconds	0.040
apscheduler	64	2700	2.965 milliseconds	3.002 milliseconds	3.006 milliseconds	0.010
apscheduler	64	10000	10.167 milliseconds	10.487 milliseconds	10.506 milliseconds	0.014
asyncio_pool	1	40	140.37 microseconds	165.56 microseconds	181.10 microseconds	0.083

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
asyncio_pool	1	300	386.47 microseconds	424.78 microseconds	434.41 microseconds	0.046
asyncio_pool	1	2700	2.752 milliseconds	2.835 milliseconds	2.987 milliseconds	0.023
asyncio_pool	1	10000	9.482 milliseconds	9.593 milliseconds	9.606 milliseconds	0.007
asyncio_pool	4	40	173.04 microseconds	188.13 microseconds	194.49 microseconds	0.098
asyncio_pool	4	300	407.48 microseconds	430.31 microseconds	434.63 microseconds	0.025
asyncio_pool	4	2700	2.755 milliseconds	2.843 milliseconds	2.872 milliseconds	0.020
asyncio_pool	4	10000	9.867 milliseconds	10.018 milliseconds	10.030 milliseconds	0.007
asyncio_pool	16	40	134.08 microseconds	145.26 microseconds	147.06 microseconds	0.048
asyncio_pool	16	300	393.00 microseconds	410.43 microseconds	413.08 microseconds	0.025
asyncio_pool	16	2700	2.802 milliseconds	2.880 milliseconds	2.901 milliseconds	0.014
asyncio_pool	16	10000	9.935 milliseconds	10.005 milliseconds	10.035 milliseconds	0.006
asyncio_pool	64	40	120.26 microseconds	127.13 microseconds	155.48 microseconds	0.075
asyncio_pool	64	300	384.40 microseconds	394.23 microseconds	419.68 microseconds	0.023
asyncio_pool	64	2700	2.811 milliseconds	2.843 milliseconds	2.879 milliseconds	0.008
asyncio_pool	64	10000	10.407 milliseconds	10.532 milliseconds	10.737 milliseconds	0.018
bare_thread	1	40	148.91 microseconds	191.86 microseconds	265.16 microseconds	0.191
bare_thread	1	300	384.68 microseconds	418.48 microseconds	436.19 microseconds	0.043
bare_thread	1	2700	2.672 milliseconds	2.738 milliseconds	2.766 milliseconds	0.011
bare_thread	1	10000	9.648 milliseconds	9.807 milliseconds	9.991 milliseconds	0.011
bare_thread	4	40	199.87 microseconds	214.91 microseconds	218.90 microseconds	0.069
bare_thread	4	300	419.42 microseconds	470.82 microseconds	476.46 microseconds	0.050
bare_thread	4	2700	2.706 milliseconds	2.777 milliseconds	2.816 milliseconds	0.013
bare_thread	4	10000	9.696 milliseconds	10.520 milliseconds	11.286 milliseconds	0.040
bare_thread	16	40	167.78 microseconds	174.20 microseconds	174.64 microseconds	0.021

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
bare_thread	16	300	424.88 microseconds	436.42 microseconds	439.49 microseconds	0.012
bare_thread	16	2700	2.711 milliseconds	2.736 milliseconds	2.749 milliseconds	0.005
bare_thread	16	10000	9.609 milliseconds	9.734 milliseconds	10.228 milliseconds	0.016
bare_thread	64	40	171.52 microseconds	175.30 microseconds	176.12 microseconds	0.015
bare_thread	64	300	424.30 microseconds	431.60 microseconds	433.02 microseconds	0.009
bare_thread	64	2700	2.675 milliseconds	2.707 milliseconds	2.771 milliseconds	0.009
bare_thread	64	10000	9.827 milliseconds	9.887 milliseconds	10.032 milliseconds	0.006
process_pool	1	40	333.53 microseconds	361.85 microseconds	366.69 microseconds	0.049
process_pool	1	300	607.37 microseconds	640.03 microseconds	647.18 microseconds	0.030
process_pool	1	2700	3.009 milliseconds	3.041 milliseconds	3.069 milliseconds	0.007
process_pool	1	10000	9.888 milliseconds	10.016 milliseconds	10.086 milliseconds	0.007
process_pool	4	40	168.78 microseconds	221.25 microseconds	235.10 microseconds	0.147
process_pool	4	300	304.29 microseconds	365.39 microseconds	376.12 microseconds	0.099
process_pool	4	2700	1.385 milliseconds	1.723 milliseconds	1.978 milliseconds	0.108
process_pool	4	10000	4.621 milliseconds	4.741 milliseconds	4.757 milliseconds	0.011
process_pool	16	40	149.92 microseconds	223.33 microseconds	232.08 microseconds	0.218
process_pool	16	300	278.51 microseconds	302.38 microseconds	334.02 microseconds	0.087
process_pool	16	2700	1.370 milliseconds	1.542 milliseconds	1.562 milliseconds	0.049
process_pool	16	10000	4.741 milliseconds	4.855 milliseconds	4.901 milliseconds	0.013
process_pool	64	40	194.08 microseconds	242.04 microseconds	250.15 microseconds	0.129
process_pool	64	300	299.76 microseconds	331.67 microseconds	337.00 microseconds	0.052
process_pool	64	2700	1.364 milliseconds	1.413 milliseconds	1.451 milliseconds	0.019
process_pool	64	10000	4.686 milliseconds	4.721 milliseconds	5.108 milliseconds	0.023
thread_pool	1	40	75.77 microseconds	96.09 microseconds	102.57 microseconds	0.119

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
thread_pool	1	300	322.77 microseconds	343.49 microseconds	347.40 microseconds	0.034
thread_pool	1	2700	2.689 milliseconds	2.738 milliseconds	2.804 milliseconds	0.012
thread_pool	1	10000	9.361 milliseconds	9.494 milliseconds	9.573 milliseconds	0.007
thread_pool	4	40	96.03 microseconds	103.48 microseconds	104.00 microseconds	0.047
thread_pool	4	300	350.77 microseconds	366.87 microseconds	371.64 microseconds	0.021
thread_pool	4	2700	2.742 milliseconds	2.787 milliseconds	2.810 milliseconds	0.013
thread_pool	4	10000	9.813 milliseconds	9.951 milliseconds	9.957 milliseconds	0.008
thread_pool	16	40	90.21 microseconds	96.78 microseconds	98.07 microseconds	0.042
thread_pool	16	300	343.15 microseconds	352.67 microseconds	355.58 microseconds	0.015
thread_pool	16	2700	2.772 milliseconds	2.806 milliseconds	2.810 milliseconds	0.009
thread_pool	16	10000	9.803 milliseconds	9.917 milliseconds	10.039 milliseconds	0.007
thread_pool	64	40	89.21 microseconds	91.13 microseconds	92.53 microseconds	0.020
thread_pool	64	300	350.35 microseconds	353.38 microseconds	355.51 microseconds	0.006
thread_pool	64	2700	2.696 milliseconds	2.730 milliseconds	2.736 milliseconds	0.008
thread_pool	64	10000	10.244 milliseconds	10.415 milliseconds	10.559 milliseconds	0.016
zoo	1	40	182.57 microseconds	223.66 microseconds	236.57 microseconds	0.111
zoo	1	300	449.17 microseconds	484.91 microseconds	497.19 microseconds	0.042
zoo	1	2700	2.754 milliseconds	2.792 milliseconds	2.799 milliseconds	0.008
zoo	1	10000	10.066 milliseconds	10.274 milliseconds	10.404 milliseconds	0.012
zoo	4	40	206.14 microseconds	237.88 microseconds	243.81 microseconds	0.075
zoo	4	300	485.26 microseconds	517.91 microseconds	527.92 microseconds	0.040
zoo	4	2700	2.946 milliseconds	3.007 milliseconds	3.025 milliseconds	0.016
zoo	4	10000	10.106 milliseconds	10.173 milliseconds	10.206 milliseconds	0.007
zoo	16	40	205.75 microseconds	235.19 microseconds	256.29 microseconds	0.072

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
zoo	16	300	470.57 microseconds	488.80 microseconds	491.20 microseconds	0.019
zoo	16	2700	2.899 milliseconds	2.938 milliseconds	2.942 milliseconds	0.010
zoo	16	10000	10.084 milliseconds	10.217 milliseconds	10.270 milliseconds	0.008
zoo	64	40	207.28 microseconds	216.84 microseconds	219.30 microseconds	0.031
zoo	64	300	491.01 microseconds	498.97 microseconds	500.32 microseconds	0.010
zoo	64	2700	2.953 milliseconds	2.994 milliseconds	2.997 milliseconds	0.008
zoo	64	10000	10.538 milliseconds	10.920 milliseconds	11.105 milliseconds	0.021

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	230.26 microseconds	79.81%	58.25 microseconds
apscheduler	1	300	243.87 microseconds	44.93%	298.89 microseconds
apscheduler	1	2700	270.38 microseconds	9.02%	2.729 milliseconds
apscheduler	1	10000	318.08 microseconds	3.19%	9.653 milliseconds
apscheduler	4	40	200.17 microseconds	78.08%	56.20 microseconds
apscheduler	4	300	202.61 microseconds	39.64%	308.46 microseconds
apscheduler	4	2700	259.91 microseconds	8.71%	2.723 milliseconds
apscheduler	4	10000	-	-	14.702 milliseconds
apscheduler	16	40	-	-	53.41 microseconds
apscheduler	16	300	-	-	319.00 microseconds
apscheduler	16	2700	-	-	2.736 milliseconds
apscheduler	16	10000	-	-	15.686 milliseconds
apscheduler	64	40	-	-	50.44 microseconds
apscheduler	64	300	-	-	306.35 microseconds
apscheduler	64	2700	-	-	2.714 milliseconds
apscheduler	64	10000	-	-	14.852 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
asyncio_pool	1	40	94.16 microseconds	67.08%	46.21 microseconds
asyncio_pool	1	300	101.41 microseconds	26.24%	285.06 microseconds
asyncio_pool	1	2700	126.14 microseconds	4.58%	2.626 milliseconds
asyncio_pool	1	10000	146.32 microseconds	1.54%	9.336 milliseconds
asyncio_pool	4	40	118.87 microseconds	68.70%	54.17 microseconds
asyncio_pool	4	300	105.33 microseconds	25.85%	302.15 microseconds
asyncio_pool	4	2700	122.61 microseconds	4.45%	2.632 milliseconds
asyncio_pool	4	10000	-	-	12.035 milliseconds
asyncio_pool	16	40	-	-	53.76 microseconds
asyncio_pool	16	300	-	-	303.75 microseconds
asyncio_pool	16	2700	-	-	2.675 milliseconds
asyncio_pool	16	10000	-	-	19.300 milliseconds
asyncio_pool	64	40	-	-	52.05 microseconds
asyncio_pool	64	300	-	-	303.49 microseconds
asyncio_pool	64	2700	-	-	2.691 milliseconds
asyncio_pool	64	10000	-	-	95.273 milliseconds
bare_thread	1	40	95.49 microseconds	64.13%	53.41 microseconds
bare_thread	1	300	97.47 microseconds	25.34%	287.21 microseconds
bare_thread	1	2700	107.34 microseconds	4.02%	2.564 milliseconds
bare_thread	1	10000	120.48 microseconds	1.25%	9.528 milliseconds
bare_thread	4	40	140.95 microseconds	70.52%	58.93 microseconds
bare_thread	4	300	118.32 microseconds	28.21%	301.10 microseconds
bare_thread	4	2700	109.53 microseconds	4.05%	2.597 milliseconds
bare_thread	4	10000	-	-	14.698 milliseconds
bare_thread	16	40	-	-	50.78 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
bare_thread	16	300	-	-	300.13 microseconds
bare_thread	16	2700	-	-	2.584 milliseconds
bare_thread	16	10000	-	-	14.647 milliseconds
bare_thread	64	40	-	-	51.21 microseconds
bare_thread	64	300	-	-	297.44 microseconds
bare_thread	64	2700	-	-	2.542 milliseconds
bare_thread	64	10000	-	-	18.793 milliseconds
process_pool	1	40	279.58 microseconds	83.82%	53.95 microseconds
process_pool	1	300	296.81 microseconds	48.87%	310.56 microseconds
process_pool	1	2700	342.13 microseconds	11.37%	2.667 milliseconds
process_pool	1	10000	341.21 microseconds	3.45%	9.547 milliseconds
process_pool	4	40	113.81 microseconds	67.43%	54.97 microseconds
process_pool	4	300	-	-	577.68 microseconds
process_pool	4	2700	-	-	4.801 milliseconds
process_pool	4	10000	-	-	17.775 milliseconds
process_pool	16	40	-	-	92.00 microseconds
process_pool	16	300	-	-	570.91 microseconds
process_pool	16	2700	-	-	9.708 milliseconds
process_pool	16	10000	-	-	54.326 milliseconds
process_pool	64	40	-	-	93.26 microseconds
process_pool	64	300	-	-	583.35 microseconds
process_pool	64	2700	-	-	27.749 milliseconds
process_pool	64	10000	-	-	173.846 milliseconds
thread_pool	1	40	31.35 microseconds	41.38%	44.42 microseconds
thread_pool	1	300	35.83 microseconds	11.10%	286.94 microseconds
thread_pool	1	2700	63.24 microseconds	2.35%	2.625 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
thread_pool	1	10000	62.33 microseconds	0.67%	9.299 milliseconds
thread_pool	4	40	46.33 microseconds	48.25%	49.70 microseconds
thread_pool	4	300	52.33 microseconds	14.92%	298.44 microseconds
thread_pool	4	2700	78.15 microseconds	2.85%	2.663 milliseconds
thread_pool	4	10000	-	-	19.045 milliseconds
thread_pool	16	40	-	-	47.32 microseconds
thread_pool	16	300	-	-	293.68 microseconds
thread_pool	16	2700	-	-	2.682 milliseconds
thread_pool	16	10000	-	-	20.052 milliseconds
thread_pool	64	40	-	-	47.07 microseconds
thread_pool	64	300	-	-	294.13 microseconds
thread_pool	64	2700	-	-	2.598 milliseconds
thread_pool	64	10000	-	-	40.559 milliseconds
zoo	1	40	130.82 microseconds	71.66%	51.75 microseconds
zoo	1	300	144.25 microseconds	32.12%	304.92 microseconds
zoo	1	2700	155.52 microseconds	5.65%	2.599 milliseconds
zoo	1	10000	256.30 microseconds	2.55%	9.810 milliseconds
zoo	4	40	151.02 microseconds	73.26%	55.12 microseconds
zoo	4	300	176.04 microseconds	36.28%	309.23 microseconds
zoo	4	2700	226.36 microseconds	7.68%	2.720 milliseconds
zoo	4	10000	-	-	15.005 milliseconds
zoo	16	40	-	-	55.32 microseconds
zoo	16	300	-	-	304.71 microseconds
zoo	16	2700	-	-	2.680 milliseconds
zoo	16	10000	-	-	19.345 milliseconds
zoo	64	40	-	-	53.22 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
zoo	64	300	-	-	314.88 microseconds
zoo	64	2700	-	-	2.722 milliseconds
zoo	64	10000	-	-	47.155 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3466.0
apscheduler	1	300	1842.4
apscheduler	1	2700	333.4
apscheduler	1	10000	100.3
apscheduler	4	40	15602.9
apscheduler	4	300	7826.8
apscheduler	4	2700	1341.0
apscheduler	4	10000	389.9
apscheduler	16	40	71424.4
apscheduler	16	300	31674.5
apscheduler	16	2700	5369.8
apscheduler	16	10000	1619.2
apscheduler	64	40	289794.7
apscheduler	64	300	127553.4
apscheduler	64	2700	21585.8
apscheduler	64	10000	6294.8
asyncio_pool	1	40	7123.9
asyncio_pool	1	300	2587.5
asyncio_pool	1	2700	363.4
asyncio_pool	1	10000	105.5
asyncio_pool	4	40	23116.5
asyncio_pool	4	300	9816.5
asyncio_pool	4	2700	1451.9
asyncio_pool	4	10000	405.4
asyncio_pool	16	40	119331.0
asyncio_pool	16	300	40712.6
asyncio_pool	16	2700	5711.1
asyncio_pool	16	10000	1610.5
asyncio_pool	64	40	532160.1
asyncio_pool	64	300	166491.4
asyncio_pool	64	2700	22767.3
asyncio_pool	64	10000	6149.5
bare_thread	1	40	6715.5
bare_thread	1	300	2599.6
bare_thread	1	2700	374.3

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
bare_thread	1	10000	103.6
bare_thread	4	40	20013.0
bare_thread	4	300	9537.0
bare_thread	4	2700	1478.1
bare_thread	4	10000	412.6
bare_thread	16	40	95364.7
bare_thread	16	300	37658.1
bare_thread	16	2700	5901.5
bare_thread	16	10000	1665.2
bare_thread	64	40	373124.9
bare_thread	64	300	150836.5
bare_thread	64	2700	23921.9
bare_thread	64	10000	6512.8
process_pool	1	40	2998.2
process_pool	1	300	1646.4
process_pool	1	2700	332.3
process_pool	1	10000	101.1
process_pool	4	40	23699.2
process_pool	4	300	13145.4
process_pool	4	2700	2888.8
process_pool	4	10000	865.5
process_pool	16	40	106727.0
process_pool	16	300	57448.9
process_pool	16	2700	11678.3
process_pool	16	10000	3375.1
process_pool	64	40	329761.1
process_pool	64	300	213501.9
process_pool	64	2700	46909.7
process_pool	64	10000	13657.9
thread_pool	1	40	13198.4
thread_pool	1	300	3098.1
thread_pool	1	2700	371.9
thread_pool	1	10000	106.8
thread_pool	4	40	41654.8
thread_pool	4	300	11403.4
thread_pool	4	2700	1459.0
thread_pool	4	10000	407.6
thread_pool	16	40	177361.2
thread_pool	16	300	46626.5
thread_pool	16	2700	5772.6
thread_pool	16	10000	1632.1
thread_pool	64	40	717389.6
thread_pool	64	300	182674.1
thread_pool	64	2700	23741.9

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
thread_pool	64	10000	6247.3
zoo	1	40	5477.4
zoo	1	300	2226.3
zoo	1	2700	363.1
zoo	1	10000	99.3
zoo	4	40	19404.4
zoo	4	300	8242.9
zoo	4	2700	1357.7
zoo	4	10000	395.8
zoo	16	40	77763.4
zoo	16	300	34001.3
zoo	16	2700	5518.7
zoo	16	10000	1586.7
zoo	64	40	308754.3
zoo	64	300	130343.7
zoo	64	2700	21671.4
zoo	64	10000	6073.5