

zoo-bench performance report

raw data: results/zoo-framework-0.7.1b0/20261011T091928Z.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)	0.7.1-beta
framework under test (module __version__)	0.7.1-beta (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==0.7.1b0

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 1.94x 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, once the body exceeds about 300 microseconds, asyncio_pool is no longer faster than the framework under test.
- 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, once the body exceeds about 10000 microseconds, asyncio_pool is no longer faster than the framework under test.
- At concurrency 1, once the body exceeds about 300 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, 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, 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, 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, once the body exceeds about 10000 microseconds, thread_pool is no longer faster than the framework under test.

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	300	this is the smallest measured tier that satisfies the condition
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	10000	this is the smallest measured tier that satisfies the condition
bare_thread	1	300	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	-	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	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	-	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	10000	this is the smallest measured tier that satisfies the condition

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	182.10 microseconds	292.58 microseconds	143.70 microseconds	149.41 microseconds	352.82 microseconds	75.72 microseconds
1	300	386.65 microseconds	549.93 microseconds	398.87 microseconds	404.88 microseconds	594.83 microseconds	330.18 microseconds
1	2700	2.736 milliseconds	2.991 milliseconds	2.778 milliseconds	2.714 milliseconds	2.951 milliseconds	2.717 milliseconds
1	10000	9.726 milliseconds	9.741 milliseconds	9.939 milliseconds	9.438 milliseconds	10.016 milliseconds	9.708 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
4	40	203.45 microseconds	264.75 microseconds	155.40 microseconds	160.45 microseconds	233.46 microseconds	96.14 microseconds
4	300	465.41 microseconds	523.52 microseconds	410.22 microseconds	421.56 microseconds	293.33 microseconds	355.71 microseconds
4	2700	2.884 milliseconds	2.993 milliseconds	2.762 milliseconds	2.677 milliseconds	1.389 milliseconds	2.729 milliseconds
4	10000	10.016 milliseconds	9.919 milliseconds	9.720 milliseconds	9.673 milliseconds	4.618 milliseconds	9.759 milliseconds
16	40	206.52 microseconds	245.94 microseconds	136.30 microseconds	170.92 microseconds	157.11 microseconds	91.05 microseconds
16	300	463.38 microseconds	491.80 microseconds	399.23 microseconds	433.71 microseconds	283.30 microseconds	346.24 microseconds
16	2700	2.875 milliseconds	2.960 milliseconds	2.844 milliseconds	2.734 milliseconds	1.430 milliseconds	2.707 milliseconds
16	10000	10.149 milliseconds	10.207 milliseconds	9.885 milliseconds	9.515 milliseconds	4.746 milliseconds	9.680 milliseconds
64	40	204.01 microseconds	224.61 microseconds	122.10 microseconds	170.82 microseconds	191.67 microseconds	90.49 microseconds
64	300	470.66 microseconds	496.93 microseconds	378.93 microseconds	427.70 microseconds	300.56 microseconds	343.57 microseconds
64	2700	2.891 milliseconds	2.961 milliseconds	2.754 milliseconds	2.687 milliseconds	1.376 milliseconds	2.766 milliseconds
64	10000	10.269 milliseconds	10.691 milliseconds	10.358 milliseconds	9.802 milliseconds	4.629 milliseconds	10.397 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	71.79%	80.15%	65.83%	64.64%	84.46%	40.53%
1	300	43.97%	44.02%	26.35%	26.00%	49.83%	12.18%
1	2700	4.94%	8.97%	4.54%	4.26%	11.47%	1.65%
1	10000	1.68%	3.67%	2.05%	1.36%	3.15%	0.70%
4	40	73.41%	79.88%	65.20%	68.16%	59.90%	48.10%
4	300	33.89%	40.02%	26.02%	29.31%	-	15.90%
4	2700	6.52%	9.78%	4.90%	4.68%	-	3.74%
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	5491.5	3417.8	6958.7	6693.0	2834.3	13207.1
1	300	2586.3	1818.4	2507.1	2469.9	1681.2	3028.6
1	2700	365.5	334.3	360.0	368.5	338.9	368.0
1	10000	102.8	102.7	100.6	106.0	99.8	103.0
4	40	19661.0	15108.5	25740.3	24929.6	17133.9	41606.5
4	300	8594.6	7640.5	9750.8	9488.5	13636.3	11245.1
4	2700	1387.0	1336.6	1448.3	1494.0	2880.3	1465.9
4	10000	399.4	403.3	411.5	413.5	866.3	409.9
16	40	77474.6	65056.7	117388.0	93609.9	101836.3	175734.6
16	300	34528.5	32533.3	40077.0	36890.8	56477.9	46210.8
16	2700	5565.2	5405.6	5625.0	5852.3	11185.9	5909.9
16	10000	1576.5	1567.5	1618.7	1681.6	3371.6	1652.9
64	40	313716.1	284933.9	524166.3	374656.8	333912.8	707233.4
64	300	135978.9	128790.1	168896.4	149637.7	212933.5	186277.5
64	2700	22138.1	21617.7	23239.8	23821.1	46518.2	23134.1
64	10000	6232.3	5986.5	6178.8	6529.1	13825.1	6155.4

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	107.03 microseconds	88.74 microseconds	51.01 microseconds	40.43 microseconds	ok
apscheduler	4	40	111.85 microseconds	536.36 microseconds	53.40 microseconds	223.21 microseconds	ok
apscheduler	1	10000	143.89 microseconds	92.82 microseconds	9.717 milliseconds	52.57 microseconds	ok
apscheduler	4	10000	115.81 microseconds	12.770 milliseconds	14.372 milliseconds	9.716 milliseconds	ok
asyncio_pool	1	40	17.44 microseconds	26.24 microseconds	49.78 microseconds	57.13 microseconds	ok
asyncio_pool	4	40	18.96 microseconds	246.77 microseconds	53.89 microseconds	277.26 microseconds	ok
asyncio_pool	1	10000	22.48 microseconds	35.28 microseconds	9.602 milliseconds	83.05 microseconds	ok
asyncio_pool	4	10000	24.14 microseconds	12.532 milliseconds	15.033 milliseconds	9.857 milliseconds	ok
bare_thread	1	40	145.72 microseconds	-77.75 microseconds	53.16 microseconds	31.65 microseconds	ok
bare_thread	4	40	152.05 microseconds	-74.78 microseconds	50.79 microseconds	328.63 microseconds	ok
bare_thread	1	10000	5.173 milliseconds	-5093.47 microseconds	9.760 milliseconds	36.84 microseconds	ok
bare_thread	4	10000	5.190 milliseconds	-5085.13 microseconds	14.764 milliseconds	10.012 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	8.19 microseconds	10.39 microseconds	44.78 microseconds	16.14 microseconds	ok
thread_pool	4	40	10.76 microseconds	188.77 microseconds	51.29 microseconds	156.66 microseconds	ok
thread_pool	1	10000	11.97 microseconds	16.35 microseconds	9.403 milliseconds	27.22 microseconds	ok
thread_pool	4	10000	13.57 microseconds	10.158 milliseconds	14.997 milliseconds	9.564 milliseconds	ok
zoo	1	40	51.83 microseconds	25.21 microseconds	53.84 microseconds	66.09 microseconds	ok
zoo	4	40	62.86 microseconds	339.07 microseconds	56.15 microseconds	317.39 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	87.64 microseconds	30.13 microseconds	9.804 milliseconds	97.83 microseconds	ok
zoo	4	10000	85.39 microseconds	10.349 milliseconds	14.876 milliseconds	10.084 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	-93.68 microseconds	return	25.66 microseconds
asyncio_pool	1	40	44.83 microseconds	submit side	34.38 microseconds
bare_thread	1	40	43.01 microseconds	handoff	102.97 microseconds
thread_pool	1	40	117.05 microseconds	return	49.95 microseconds
apscheduler	4	40	-169.63 microseconds	return	94.17 microseconds
asyncio_pool	4	40	147.56 microseconds	handoff	92.30 microseconds
bare_thread	4	40	287.24 microseconds	handoff	413.84 microseconds
thread_pool	4	40	329.76 microseconds	return	160.73 microseconds
apscheduler	1	10000	6.76 microseconds	body	87.12 microseconds
asyncio_pool	1	10000	249.94 microseconds	body	202.42 microseconds
bare_thread	1	10000	136.37 microseconds	handoff	5.124 milliseconds
thread_pool	1	10000	550.56 microseconds	body	401.60 microseconds
apscheduler	4	10000	521.17 microseconds	body	503.56 microseconds
asyncio_pool	4	10000	-75.24 microseconds	return	226.80 microseconds
bare_thread	4	10000	8.602 milliseconds	handoff	15.434 milliseconds
thread_pool	4	10000	552.31 microseconds	return	520.56 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

191.01 microseconds vs with 196.76 microseconds, a difference of 5.75 microseconds; zoo 40 microseconds /

concurrency 4: without instrumentation 753.54 microseconds vs with 744.51 microseconds, a difference of 9.02 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 1: without instrumentation 10.042 milliseconds vs with 10.014 milliseconds, a difference of 28.37 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 10000 microseconds / concurrency 4: without instrumentation 40.527 milliseconds vs with 40.236 milliseconds, a difference of 290.99 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 4 overlaps 61.2%; zoo 10000 microseconds / concurrency 1 overlaps 16.2%; 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) 2.12 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 25.01 microseconds; per-round policy lookups (period / phase / timeout each read config) 17.20 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 7.39 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) 36.22 microseconds; per-round policy lookups (period / phase / timeout each read config) 31.27 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.08 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 3.16 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 40.73 microseconds; per-round policy lookups (period / phase / timeout each read config) 29.73 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 13.49 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) 44.37 microseconds; per-round policy lookups (period / phase / timeout each read config) 34.62 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 6.55 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-94 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 45 microseconds more per task than `asyncio_pool`, most of it on 'submit side' (+34 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 43 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 117 microseconds more per task than `thread_pool`, most of it on 'return' (+50 microseconds).

- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 2, dispatch (handed to the scheduling model, incl. pool enqueue) 25, per-round policy lookups (period / phase / timeout each read config) 17, submit side, other (the adapter's own bookkeeping and completion-signal state) 7.
- At concurrency 4 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-170 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 148 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+92 microseconds).
- At concurrency 4 and a 40 microsecond body, the framework under test spends 287 microseconds more per task than `bare_thread`, most of it on 'handoff' (+414 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -89 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 330 microseconds more per task than `thread_pool`, most of it on 'return' (+161 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) 36, per-round policy lookups (period / phase / timeout each read config) 31, 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 spends 7 microseconds more per task than `apscheduler`, most of it on 'body' (+87 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -63 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 250 microseconds more per task than `asyncio_pool`, most of it on 'body' (+202 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -5 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 136 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5124 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5085 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 551 microseconds more per task than `thread_pool`, most of it on 'body' (+402 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 3, dispatch (handed to the scheduling model, incl. pool enqueue) 41, per-round policy lookups (period / phase / timeout each read config) 30, submit side, other (the adapter's own bookkeeping and completion-signal state) 13.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 521 microseconds more per task than `apscheduler`, most of it on 'body' (+504 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -2422 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test is not slower overall than `asyncio_pool` (-75 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 8602 microseconds more per task than bare_thread, most of it on 'handoff' (+15434 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5105 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 552 microseconds more per task than thread_pool, most of it on 'return' (+521 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -121 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) 35, submit side, other (the adapter's own bookkeeping and completion-signal state) 7.

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:365; 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 1.61x faster than apscheduler	182.10 microseconds
process_pool	1	40	the framework under test is 1.94x faster than process_pool	182.10 microseconds
apscheduler	1	300	the framework under test is 1.42x faster than apscheduler	386.65 microseconds
process_pool	1	300	the framework under test is 1.54x faster than process_pool	386.65 microseconds
apscheduler	1	2700	the framework under test is 1.09x faster than apscheduler	2.736 milliseconds
process_pool	1	2700	the framework under test is 1.08x faster than process_pool	2.736 milliseconds
asyncio_pool	1	10000	the framework under test is 1.02x faster than asyncio_pool	9.726 milliseconds
process_pool	1	10000	the framework under test is 1.03x faster than process_pool	9.726 milliseconds
apscheduler	4	40	the framework under test is 1.30x faster than apscheduler	203.45 microseconds
apscheduler	4	300	the framework under test is 1.12x faster than apscheduler	465.41 microseconds
apscheduler	4	2700	the framework under test is 1.04x faster than apscheduler	2.884 milliseconds
apscheduler	16	40	the framework under test is 1.19x faster than apscheduler	206.52 microseconds
apscheduler	16	300	the framework under test is 1.06x faster than apscheduler	463.38 microseconds
apscheduler	16	2700	the framework under test is 1.03x faster than apscheduler	2.875 milliseconds
apscheduler	64	300	the framework under test is 1.06x faster than apscheduler	470.66 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
asyncio_pool	1	40	asyncio_pool is 1.27x faster than zoo	182.10 microseconds
thread_pool	1	40	thread_pool is 2.41x faster than zoo	182.10 microseconds
bare_thread	1	10000	bare_thread is 1.03x faster than zoo	9.726 milliseconds
asyncio_pool	4	40	asyncio_pool is 1.31x faster than zoo	203.45 microseconds
bare_thread	4	40	bare_thread is 1.27x faster than zoo	203.45 microseconds
thread_pool	4	40	thread_pool is 2.12x faster than zoo	203.45 microseconds
asyncio_pool	4	300	asyncio_pool is 1.13x faster than zoo	465.41 microseconds
bare_thread	4	300	bare_thread is 1.10x faster than zoo	465.41 microseconds
process_pool	4	300	process_pool is 1.59x faster than zoo	465.41 microseconds
thread_pool	4	300	thread_pool is 1.31x faster than zoo	465.41 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.04x faster than zoo	2.884 milliseconds
bare_thread	4	2700	bare_thread is 1.08x faster than zoo	2.884 milliseconds
process_pool	4	2700	process_pool is 2.08x faster than zoo	2.884 milliseconds
thread_pool	4	2700	thread_pool is 1.06x faster than zoo	2.884 milliseconds
asyncio_pool	4	10000	asyncio_pool is 1.03x faster than zoo	10.016 milliseconds
bare_thread	4	10000	bare_thread is 1.04x faster than zoo	10.016 milliseconds
process_pool	4	10000	process_pool is 2.17x faster than zoo	10.016 milliseconds
thread_pool	4	10000	thread_pool is 1.03x faster than zoo	10.016 milliseconds
asyncio_pool	16	40	asyncio_pool is 1.52x faster than zoo	206.52 microseconds
bare_thread	16	40	bare_thread is 1.21x faster than zoo	206.52 microseconds
process_pool	16	40	process_pool is 1.31x faster than zoo	206.52 microseconds
thread_pool	16	40	thread_pool is 2.27x faster than zoo	206.52 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
asyncio_pool	16	300	asyncio_pool is 1.16x faster than zoo	463.38 microseconds
bare_thread	16	300	bare_thread is 1.07x faster than zoo	463.38 microseconds
process_pool	16	300	process_pool is 1.64x faster than zoo	463.38 microseconds
thread_pool	16	300	thread_pool is 1.34x faster than zoo	463.38 microseconds
bare_thread	16	2700	bare_thread is 1.05x faster than zoo	2.875 milliseconds
process_pool	16	2700	process_pool is 2.01x faster than zoo	2.875 milliseconds
thread_pool	16	2700	thread_pool is 1.06x faster than zoo	2.875 milliseconds
asyncio_pool	16	10000	asyncio_pool is 1.03x faster than zoo	10.149 milliseconds
bare_thread	16	10000	bare_thread is 1.07x faster than zoo	10.149 milliseconds
process_pool	16	10000	process_pool is 2.14x faster than zoo	10.149 milliseconds
thread_pool	16	10000	thread_pool is 1.05x faster than zoo	10.149 milliseconds
asyncio_pool	64	40	asyncio_pool is 1.67x faster than zoo	204.01 microseconds
bare_thread	64	40	bare_thread is 1.19x faster than zoo	204.01 microseconds
thread_pool	64	40	thread_pool is 2.25x faster than zoo	204.01 microseconds
asyncio_pool	64	300	asyncio_pool is 1.24x faster than zoo	470.66 microseconds
bare_thread	64	300	bare_thread is 1.10x faster than zoo	470.66 microseconds
process_pool	64	300	process_pool is 1.57x faster than zoo	470.66 microseconds
thread_pool	64	300	thread_pool is 1.37x faster than zoo	470.66 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.05x faster than zoo	2.891 milliseconds
bare_thread	64	2700	bare_thread is 1.08x faster than zoo	2.891 milliseconds
process_pool	64	2700	process_pool is 2.10x faster than zoo	2.891 milliseconds
thread_pool	64	2700	thread_pool is 1.04x faster than zoo	2.891 milliseconds
bare_thread	64	10000	bare_thread is 1.05x faster than zoo	10.269 milliseconds
process_pool	64	10000	process_pool is 2.22x faster than zoo	10.269 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: 0.9% to 24.7% (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 18.0%, less than the band of 20.5% - no winner
asyncio_pool	1	300	1.03x	the two sides differ by 3.2%, less than the band of 18.0% - no winner
bare_thread	1	300	1.05x	the two sides differ by 4.7%, less than the band of 17.5% - no winner
thread_pool	1	300	0.85x	the two sides differ by 14.6%, less than the band of 16.8% - no winner
asyncio_pool	1	2700	1.02x	the two sides differ by 1.5%, less than the band of 3.3% - no winner
bare_thread	1	2700	0.99x	the two sides differ by 0.8%, less than the band of 2.5% - no winner
thread_pool	1	2700	0.99x	the two sides differ by 0.7%, less than the band of 1.5% - no winner
apscheduler	1	10000	1.00x	the two sides differ by 0.2%, less than the band of 1.1% - no winner
thread_pool	1	10000	1.00x	the two sides differ by 0.2%, less than the band of 1.7% - no winner
process_pool	4	40	1.15x	the two sides differ by 14.7%, less than the band of 21.8% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
apscheduler	4	10000	0.99x	the two sides differ by 1.0%, less than the band of 5.0% - no winner
asyncio_pool	16	2700	0.99x	the two sides differ by 1.1%, less than the band of 5.0% - no winner
apscheduler	16	10000	1.01x	the two sides differ by 0.6%, less than the band of 1.6% - no winner
apscheduler	64	40	1.10x	the two sides differ by 10.1%, less than the band of 16.6% - no winner
process_pool	64	40	0.94x	the two sides differ by 6.0%, less than the band of 12.0% - no winner
apscheduler	64	2700	1.02x	the two sides differ by 2.4%, less than the band of 2.9% - no winner
apscheduler	64	10000	1.04x	the two sides differ by 4.1%, less than the band of 4.9% - no winner
asyncio_pool	64	10000	1.01x	the two sides differ by 0.9%, less than the band of 3.8% - no winner
thread_pool	64	10000	1.01x	the two sides differ by 1.3%, 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': 16507, 'child_pids': [16965, 16968, 16971, 16974, 16977, 16983, 16989, 16995, 17001, 17019, 17037, 17055, 17076, 17132, 17187, 17241, 17308, 17310, 17312, 17314, 17316, 17321, 17326, 17331, 17336, 17353, 17370, 17388, 17405, 17464, 17523, 17585, 17654, 17688, 17722, 17756, 17790, 17923, 18056, 18189, 18322, 18851, 19380, 19909, 20439, 22552, 24665, 26778, 28894, 28898, 28902, 28906, 28910, 28917, 28924, 28931, 28938, 28957, 28976, 28995, 29014, 29081, 29148, 29215, 29284, 29286, 29288, 29290, 29292, 29297, 29302, 29307, 29312, 29329, 29346, 29363, 29380, 29437, 29494, 29552, 29618, 29620, 29622, 29624, 29626, 29631, 29636, 29641, 29646, 29663, 29680, 29699, 29716, 29753, 29789, 29830], '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	292.58 microseconds	336.06 microseconds	345.13 microseconds	0.065
apscheduler	1	300	549.93 microseconds	599.28 microseconds	604.80 microseconds	0.038
apscheduler	1	2700	2.991 milliseconds	3.029 milliseconds	3.303 milliseconds	0.033
apscheduler	1	10000	9.741 milliseconds	9.805 milliseconds	9.851 milliseconds	0.004
apscheduler	4	40	264.75 microseconds	271.06 microseconds	293.70 microseconds	0.031
apscheduler	4	300	523.52 microseconds	548.69 microseconds	583.69 microseconds	0.029
apscheduler	4	2700	2.993 milliseconds	3.061 milliseconds	3.107 milliseconds	0.013
apscheduler	4	10000	9.919 milliseconds	10.071 milliseconds	11.589 milliseconds	0.043
apscheduler	16	40	245.94 microseconds	257.54 microseconds	259.33 microseconds	0.028
apscheduler	16	300	491.80 microseconds	525.31 microseconds	537.30 microseconds	0.032
apscheduler	16	2700	2.960 milliseconds	3.070 milliseconds	3.141 milliseconds	0.018
apscheduler	16	10000	10.207 milliseconds	10.331 milliseconds	10.404 milliseconds	0.011
apscheduler	64	40	224.61 microseconds	311.15 microseconds	313.64 microseconds	0.133
apscheduler	64	300	496.93 microseconds	541.31 microseconds	550.31 microseconds	0.035
apscheduler	64	2700	2.961 milliseconds	3.037 milliseconds	3.203 milliseconds	0.022

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	64	10000	10.691 milliseconds	10.787 milliseconds	10.808 milliseconds	0.026
asyncio_pool	1	40	143.70 microseconds	168.43 microseconds	173.46 microseconds	0.071
asyncio_pool	1	300	398.87 microseconds	434.65 microseconds	468.32 microseconds	0.049
asyncio_pool	1	2700	2.778 milliseconds	2.871 milliseconds	3.029 milliseconds	0.025
asyncio_pool	1	10000	9.939 milliseconds	10.037 milliseconds	10.061 milliseconds	0.006
asyncio_pool	4	40	155.40 microseconds	183.35 microseconds	187.06 microseconds	0.088
asyncio_pool	4	300	410.22 microseconds	424.86 microseconds	436.27 microseconds	0.026
asyncio_pool	4	2700	2.762 milliseconds	2.837 milliseconds	2.858 milliseconds	0.014
asyncio_pool	4	10000	9.720 milliseconds	9.864 milliseconds	10.027 milliseconds	0.010
asyncio_pool	16	40	136.30 microseconds	154.75 microseconds	161.27 microseconds	0.079
asyncio_pool	16	300	399.23 microseconds	412.12 microseconds	415.30 microseconds	0.023
asyncio_pool	16	2700	2.844 milliseconds	3.071 milliseconds	3.292 milliseconds	0.040
asyncio_pool	16	10000	9.885 milliseconds	10.012 milliseconds	10.668 milliseconds	0.021
asyncio_pool	64	40	122.10 microseconds	125.76 microseconds	127.15 microseconds	0.019
asyncio_pool	64	300	378.93 microseconds	385.83 microseconds	386.46 microseconds	0.011
asyncio_pool	64	2700	2.754 milliseconds	2.796 milliseconds	2.860 milliseconds	0.011
asyncio_pool	64	10000	10.358 milliseconds	10.465 milliseconds	10.487 milliseconds	0.015
bare_thread	1	40	149.41 microseconds	189.01 microseconds	193.86 microseconds	0.126
bare_thread	1	300	404.88 microseconds	445.16 microseconds	451.81 microseconds	0.044
bare_thread	1	2700	2.714 milliseconds	2.798 milliseconds	2.878 milliseconds	0.017
bare_thread	1	10000	9.438 milliseconds	9.589 milliseconds	9.611 milliseconds	0.007
bare_thread	4	40	160.45 microseconds	178.72 microseconds	191.71 microseconds	0.057
bare_thread	4	300	421.56 microseconds	440.96 microseconds	448.37 microseconds	0.025
bare_thread	4	2700	2.677 milliseconds	2.733 milliseconds	2.753 milliseconds	0.009

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
bare_thread	4	10000	9.673 milliseconds	9.764 milliseconds	10.007 milliseconds	0.010
bare_thread	16	40	170.92 microseconds	179.62 microseconds	180.11 microseconds	0.026
bare_thread	16	300	433.71 microseconds	447.00 microseconds	450.27 microseconds	0.014
bare_thread	16	2700	2.734 milliseconds	2.765 milliseconds	2.767 milliseconds	0.005
bare_thread	16	10000	9.515 milliseconds	9.597 milliseconds	9.637 milliseconds	0.004
bare_thread	64	40	170.82 microseconds	173.53 microseconds	174.01 microseconds	0.009
bare_thread	64	300	427.70 microseconds	447.06 microseconds	468.46 microseconds	0.024
bare_thread	64	2700	2.687 milliseconds	2.778 milliseconds	2.947 milliseconds	0.023
bare_thread	64	10000	9.802 milliseconds	9.963 milliseconds	10.025 milliseconds	0.008
process_pool	1	40	352.82 microseconds	384.48 microseconds	393.24 microseconds	0.057
process_pool	1	300	594.83 microseconds	632.73 microseconds	767.68 microseconds	0.075
process_pool	1	2700	2.951 milliseconds	2.990 milliseconds	3.001 milliseconds	0.008
process_pool	1	10000	10.016 milliseconds	10.125 milliseconds	10.169 milliseconds	0.005
process_pool	4	40	233.46 microseconds	270.06 microseconds	288.01 microseconds	0.139
process_pool	4	300	293.33 microseconds	348.68 microseconds	351.02 microseconds	0.072
process_pool	4	2700	1.389 milliseconds	1.500 milliseconds	1.547 milliseconds	0.034
process_pool	4	10000	4.618 milliseconds	4.788 milliseconds	4.890 milliseconds	0.016
process_pool	16	40	157.11 microseconds	215.42 microseconds	223.27 microseconds	0.180
process_pool	16	300	283.30 microseconds	348.25 microseconds	362.65 microseconds	0.103
process_pool	16	2700	1.430 milliseconds	1.563 milliseconds	1.599 milliseconds	0.043
process_pool	16	10000	4.746 milliseconds	4.866 milliseconds	4.892 milliseconds	0.018
process_pool	64	40	191.67 microseconds	226.49 microseconds	236.11 microseconds	0.088
process_pool	64	300	300.56 microseconds	334.45 microseconds	347.78 microseconds	0.057
process_pool	64	2700	1.376 milliseconds	1.401 milliseconds	1.413 milliseconds	0.012

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
process_pool	64	10000	4.629 milliseconds	4.715 milliseconds	4.881 milliseconds	0.014
thread_pool	1	40	75.72 microseconds	100.80 microseconds	121.94 microseconds	0.168
thread_pool	1	300	330.18 microseconds	350.74 microseconds	362.26 microseconds	0.037
thread_pool	1	2700	2.717 milliseconds	2.763 milliseconds	2.769 milliseconds	0.007
thread_pool	1	10000	9.708 milliseconds	9.862 milliseconds	10.024 milliseconds	0.010
thread_pool	4	40	96.14 microseconds	103.84 microseconds	104.87 microseconds	0.040
thread_pool	4	300	355.71 microseconds	370.90 microseconds	376.79 microseconds	0.022
thread_pool	4	2700	2.729 milliseconds	2.823 milliseconds	2.839 milliseconds	0.018
thread_pool	4	10000	9.759 milliseconds	9.887 milliseconds	9.994 milliseconds	0.009
thread_pool	16	40	91.05 microseconds	94.68 microseconds	97.78 microseconds	0.032
thread_pool	16	300	346.24 microseconds	358.28 microseconds	362.52 microseconds	0.018
thread_pool	16	2700	2.707 milliseconds	2.746 milliseconds	2.754 milliseconds	0.009
thread_pool	16	10000	9.680 milliseconds	9.832 milliseconds	9.950 milliseconds	0.009
thread_pool	64	40	90.49 microseconds	92.03 microseconds	92.25 microseconds	0.015
thread_pool	64	300	343.57 microseconds	347.83 microseconds	351.25 microseconds	0.008
thread_pool	64	2700	2.766 milliseconds	2.799 milliseconds	2.822 milliseconds	0.008
thread_pool	64	10000	10.397 milliseconds	10.501 milliseconds	10.626 milliseconds	0.013
zoo	1	40	182.10 microseconds	212.72 microseconds	219.88 microseconds	0.079
zoo	1	300	386.65 microseconds	493.90 microseconds	501.64 microseconds	0.130
zoo	1	2700	2.736 milliseconds	2.784 milliseconds	2.814 milliseconds	0.008
zoo	1	10000	9.726 milliseconds	9.847 milliseconds	9.885 milliseconds	0.007
zoo	4	40	203.45 microseconds	213.52 microseconds	256.87 microseconds	0.079
zoo	4	300	465.41 microseconds	494.09 microseconds	521.87 microseconds	0.039
zoo	4	2700	2.884 milliseconds	2.956 milliseconds	2.961 milliseconds	0.014

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
zoo	4	10000	10.016 milliseconds	10.137 milliseconds	10.200 milliseconds	0.007
zoo	16	40	206.52 microseconds	213.77 microseconds	214.47 microseconds	0.029
zoo	16	300	463.38 microseconds	473.35 microseconds	475.28 microseconds	0.015
zoo	16	2700	2.875 milliseconds	2.913 milliseconds	2.922 milliseconds	0.010
zoo	16	10000	10.149 milliseconds	10.221 milliseconds	10.256 milliseconds	0.005
zoo	64	40	204.01 microseconds	208.81 microseconds	225.33 microseconds	0.032
zoo	64	300	470.66 microseconds	477.84 microseconds	479.97 microseconds	0.009
zoo	64	2700	2.891 milliseconds	2.931 milliseconds	2.941 milliseconds	0.007
zoo	64	10000	10.269 milliseconds	10.682 milliseconds	10.891 milliseconds	0.024

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	234.51 microseconds	80.15%	58.08 microseconds
apscheduler	1	300	242.09 microseconds	44.02%	307.83 microseconds
apscheduler	1	2700	268.18 microseconds	8.97%	2.723 milliseconds
apscheduler	1	10000	357.36 microseconds	3.67%	9.384 milliseconds
apscheduler	4	40	211.48 microseconds	79.88%	53.27 microseconds
apscheduler	4	300	209.50 microseconds	40.02%	314.03 microseconds
apscheduler	4	2700	292.59 microseconds	9.78%	2.700 milliseconds
apscheduler	4	10000	-	-	13.824 milliseconds
apscheduler	16	40	-	-	52.81 microseconds
apscheduler	16	300	-	-	309.74 microseconds
apscheduler	16	2700	-	-	2.710 milliseconds
apscheduler	16	10000	-	-	19.774 milliseconds
apscheduler	64	40	-	-	50.68 microseconds
apscheduler	64	300	-	-	306.77 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	64	2700	-	-	2.697 milliseconds
apscheduler	64	10000	-	-	26.236 milliseconds
asyncio_pool	1	40	94.60 microseconds	65.83%	49.11 microseconds
asyncio_pool	1	300	105.10 microseconds	26.35%	293.77 microseconds
asyncio_pool	1	2700	126.11 microseconds	4.54%	2.652 milliseconds
asyncio_pool	1	10000	203.48 microseconds	2.05%	9.735 milliseconds
asyncio_pool	4	40	101.32 microseconds	65.20%	54.08 microseconds
asyncio_pool	4	300	106.75 microseconds	26.02%	303.47 microseconds
asyncio_pool	4	2700	135.20 microseconds	4.90%	2.627 milliseconds
asyncio_pool	4	10000	-	-	11.893 milliseconds
asyncio_pool	16	40	-	-	54.43 microseconds
asyncio_pool	16	300	-	-	305.14 microseconds
asyncio_pool	16	2700	-	-	2.720 milliseconds
asyncio_pool	16	10000	-	-	19.853 milliseconds
asyncio_pool	64	40	-	-	52.71 microseconds
asyncio_pool	64	300	-	-	296.74 microseconds
asyncio_pool	64	2700	-	-	2.635 milliseconds
asyncio_pool	64	10000	-	-	95.672 milliseconds
bare_thread	1	40	96.58 microseconds	64.64%	52.83 microseconds
bare_thread	1	300	105.26 microseconds	26.00%	299.62 microseconds
bare_thread	1	2700	115.55 microseconds	4.26%	2.598 milliseconds
bare_thread	1	10000	128.17 microseconds	1.36%	9.310 milliseconds
bare_thread	4	40	109.36 microseconds	68.16%	51.09 microseconds
bare_thread	4	300	123.57 microseconds	29.31%	297.99 microseconds
bare_thread	4	2700	125.35 microseconds	4.68%	2.552 milliseconds
bare_thread	4	10000	-	-	14.624 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
bare_thread	16	40	-	-	51.20 microseconds
bare_thread	16	300	-	-	302.31 microseconds
bare_thread	16	2700	-	-	2.600 milliseconds
bare_thread	16	10000	-	-	14.517 milliseconds
bare_thread	64	40	-	-	51.67 microseconds
bare_thread	64	300	-	-	296.28 microseconds
bare_thread	64	2700	-	-	2.547 milliseconds
bare_thread	64	10000	-	-	14.926 milliseconds
process_pool	1	40	297.99 microseconds	84.46%	54.82 microseconds
process_pool	1	300	296.42 microseconds	49.83%	298.41 microseconds
process_pool	1	2700	338.48 microseconds	11.47%	2.612 milliseconds
process_pool	1	10000	315.65 microseconds	3.15%	9.700 milliseconds
process_pool	4	40	139.83 microseconds	59.90%	93.63 microseconds
process_pool	4	300	-	-	570.65 microseconds
process_pool	4	2700	-	-	4.811 milliseconds
process_pool	4	10000	-	-	17.796 milliseconds
process_pool	16	40	-	-	89.18 microseconds
process_pool	16	300	-	-	566.98 microseconds
process_pool	16	2700	-	-	9.350 milliseconds
process_pool	16	10000	-	-	52.201 milliseconds
process_pool	64	40	-	-	90.02 microseconds
process_pool	64	300	-	-	578.40 microseconds
process_pool	64	2700	-	-	27.348 milliseconds
process_pool	64	10000	-	-	158.515 milliseconds
thread_pool	1	40	30.69 microseconds	40.53%	45.03 microseconds
thread_pool	1	300	40.23 microseconds	12.18%	289.95 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
thread_pool	1	2700	44.78 microseconds	1.65%	2.672 milliseconds
thread_pool	1	10000	68.09 microseconds	0.70%	9.640 milliseconds
thread_pool	4	40	46.24 microseconds	48.10%	49.90 microseconds
thread_pool	4	300	56.54 microseconds	15.90%	299.17 microseconds
thread_pool	4	2700	101.92 microseconds	3.74%	2.627 milliseconds
thread_pool	4	10000	-	-	19.739 milliseconds
thread_pool	16	40	-	-	47.80 microseconds
thread_pool	16	300	-	-	294.69 microseconds
thread_pool	16	2700	-	-	2.611 milliseconds
thread_pool	16	10000	-	-	19.834 milliseconds
thread_pool	64	40	-	-	47.57 microseconds
thread_pool	64	300	-	-	286.81 microseconds
thread_pool	64	2700	-	-	2.685 milliseconds
thread_pool	64	10000	-	-	73.761 milliseconds
zoo	1	40	130.73 microseconds	71.79%	51.38 microseconds
zoo	1	300	170.02 microseconds	43.97%	216.63 microseconds
zoo	1	2700	135.03 microseconds	4.94%	2.601 milliseconds
zoo	1	10000	163.30 microseconds	1.68%	9.563 milliseconds
zoo	4	40	149.36 microseconds	73.41%	54.09 microseconds
zoo	4	300	157.75 microseconds	33.89%	307.66 microseconds
zoo	4	2700	188.10 microseconds	6.52%	2.696 milliseconds
zoo	4	10000	-	-	14.842 milliseconds
zoo	16	40	-	-	53.93 microseconds
zoo	16	300	-	-	304.16 microseconds
zoo	16	2700	-	-	2.656 milliseconds
zoo	16	10000	-	-	20.061 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
zoo	64	40	-	-	53.82 microseconds
zoo	64	300	-	-	308.19 microseconds
zoo	64	2700	-	-	2.676 milliseconds
zoo	64	10000	-	-	31.988 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3417.8
apscheduler	1	300	1818.4
apscheduler	1	2700	334.3
apscheduler	1	10000	102.7
apscheduler	4	40	15108.5
apscheduler	4	300	7640.5
apscheduler	4	2700	1336.6
apscheduler	4	10000	403.3
apscheduler	16	40	65056.7
apscheduler	16	300	32533.3
apscheduler	16	2700	5405.6
apscheduler	16	10000	1567.5
apscheduler	64	40	284933.9
apscheduler	64	300	128790.1
apscheduler	64	2700	21617.7
apscheduler	64	10000	5986.5
asyncio_pool	1	40	6958.7
asyncio_pool	1	300	2507.1
asyncio_pool	1	2700	360.0
asyncio_pool	1	10000	100.6
asyncio_pool	4	40	25740.3
asyncio_pool	4	300	9750.8
asyncio_pool	4	2700	1448.3
asyncio_pool	4	10000	411.5
asyncio_pool	16	40	117388.0
asyncio_pool	16	300	40077.0
asyncio_pool	16	2700	5625.0
asyncio_pool	16	10000	1618.7
asyncio_pool	64	40	524166.3
asyncio_pool	64	300	168896.4
asyncio_pool	64	2700	23239.8
asyncio_pool	64	10000	6178.8
bare_thread	1	40	6693.0
bare_thread	1	300	2469.9

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
bare_thread	1	2700	368.5
bare_thread	1	10000	106.0
bare_thread	4	40	24929.6
bare_thread	4	300	9488.5
bare_thread	4	2700	1494.0
bare_thread	4	10000	413.5
bare_thread	16	40	93609.9
bare_thread	16	300	36890.8
bare_thread	16	2700	5852.3
bare_thread	16	10000	1681.6
bare_thread	64	40	374656.8
bare_thread	64	300	149637.7
bare_thread	64	2700	23821.1
bare_thread	64	10000	6529.1
process_pool	1	40	2834.3
process_pool	1	300	1681.2
process_pool	1	2700	338.9
process_pool	1	10000	99.8
process_pool	4	40	17133.9
process_pool	4	300	13636.3
process_pool	4	2700	2880.3
process_pool	4	10000	866.3
process_pool	16	40	101836.3
process_pool	16	300	56477.9
process_pool	16	2700	11185.9
process_pool	16	10000	3371.6
process_pool	64	40	333912.8
process_pool	64	300	212933.5
process_pool	64	2700	46518.2
process_pool	64	10000	13825.1
thread_pool	1	40	13207.1
thread_pool	1	300	3028.6
thread_pool	1	2700	368.0
thread_pool	1	10000	103.0
thread_pool	4	40	41606.5
thread_pool	4	300	11245.1
thread_pool	4	2700	1465.9
thread_pool	4	10000	409.9
thread_pool	16	40	175734.6
thread_pool	16	300	46210.8
thread_pool	16	2700	5909.9
thread_pool	16	10000	1652.9
thread_pool	64	40	707233.4
thread_pool	64	300	186277.5

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
thread_pool	64	2700	23134.1
thread_pool	64	10000	6155.4
zoo	1	40	5491.5
zoo	1	300	2586.3
zoo	1	2700	365.5
zoo	1	10000	102.8
zoo	4	40	19661.0
zoo	4	300	8594.6
zoo	4	2700	1387.0
zoo	4	10000	399.4
zoo	16	40	77474.6
zoo	16	300	34528.5
zoo	16	2700	5565.2
zoo	16	10000	1576.5
zoo	64	40	313716.1
zoo	64	300	135978.9
zoo	64	2700	22138.1
zoo	64	10000	6232.3