

zoo-bench performance report

raw data: results/zoo-framework-0.8.0/20261011T092340Z.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)	0.8.0
framework under test (module __version__)	0.8.0 (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==0.8.0

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

Workload

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

Measurement methodology

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

Conclusion summary

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

- At concurrency 1 and a 40 microsecond body, the framework under test is 2.12x 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 40 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, 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, once the body exceeds about 40 microseconds, process_pool is no longer faster than the framework under test.
- At concurrency 1, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers thread_pool is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.

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

millisecond one)

Overhead threshold crossings

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

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

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

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

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

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

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

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

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

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

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

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

Relative turning points

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

Charts

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

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

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

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

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

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

Dimension: Latency percentiles and jitter

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

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	174.06 microseconds	281.24 microseconds	213.36 microseconds	144.76 microseconds	369.75 microseconds	77.28 microseconds
1	300	433.22 microseconds	544.52 microseconds	405.28 microseconds	396.70 microseconds	614.59 microseconds	324.43 microseconds
1	2700	2.800 milliseconds	2.884 milliseconds	2.701 milliseconds	2.680 milliseconds	2.982 milliseconds	2.658 milliseconds
1	10000	9.885 milliseconds	10.156 milliseconds	9.700 milliseconds	9.548 milliseconds	10.087 milliseconds	9.698 milliseconds

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
4	40	200.10 microseconds	255.42 microseconds	156.05 microseconds	168.63 microseconds	191.30 microseconds	96.24 microseconds
4	300	457.43 microseconds	516.26 microseconds	407.59 microseconds	412.07 microseconds	341.25 microseconds	352.35 microseconds
4	2700	2.814 milliseconds	2.983 milliseconds	2.764 milliseconds	2.728 milliseconds	1.359 milliseconds	2.711 milliseconds
4	10000	10.050 milliseconds	10.101 milliseconds	9.786 milliseconds	9.829 milliseconds	4.650 milliseconds	9.811 milliseconds
16	40	199.72 microseconds	246.53 microseconds	130.72 microseconds	170.14 microseconds	152.53 microseconds	88.97 microseconds
16	300	464.10 microseconds	520.81 microseconds	392.41 microseconds	419.21 microseconds	277.10 microseconds	354.95 microseconds
16	2700	2.869 milliseconds	2.998 milliseconds	2.756 milliseconds	2.682 milliseconds	1.413 milliseconds	2.133 milliseconds
16	10000	10.104 milliseconds	10.223 milliseconds	9.931 milliseconds	9.895 milliseconds	4.698 milliseconds	9.734 milliseconds
64	40	195.85 microseconds	245.59 microseconds	123.00 microseconds	174.50 microseconds	200.04 microseconds	90.65 microseconds
64	300	465.14 microseconds	514.29 microseconds	381.41 microseconds	426.28 microseconds	310.51 microseconds	353.92 microseconds
64	2700	2.881 milliseconds	2.940 milliseconds	2.804 milliseconds	2.775 milliseconds	1.375 milliseconds	2.774 milliseconds
64	10000	10.631 milliseconds	9.958 milliseconds	10.397 milliseconds	9.824 milliseconds	4.637 milliseconds	10.331 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	71.17%	82.72%	69.72%	63.87%	84.78%	39.13%
1	300	30.28%	45.98%	26.53%	25.54%	49.47%	11.12%
1	2700	4.86%	10.26%	4.58%	4.26%	11.34%	1.81%
1	10000	1.50%	3.60%	1.58%	1.33%	3.39%	0.73%
4	40	72.44%	78.18%	65.64%	69.06%	53.06%	48.39%
4	300	33.04%	40.22%	25.72%	28.67%	-	15.31%
4	2700	7.13%	9.79%	4.02%	4.69%	-	3.67%
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	5745.3	3555.7	4686.8	6907.9	2704.5	12939.1
1	300	2308.3	1836.5	2467.4	2520.8	1627.1	3082.3
1	2700	357.2	346.7	370.2	373.2	335.3	376.2
1	10000	101.2	98.5	103.1	104.7	99.1	103.1
4	40	19989.9	15660.3	25632.9	23720.3	20909.5	41562.7
4	300	8744.6	7748.0	9813.8	9707.1	11721.5	11352.2
4	2700	1421.5	1340.8	1447.4	1466.1	2944.3	1475.5
4	10000	398.0	396.0	408.8	407.0	860.2	407.7
16	40	80111.5	64899.7	122396.1	94038.7	104898.1	179841.5
16	300	34475.3	30721.3	40773.2	38167.2	57740.1	45076.5
16	2700	5576.0	5337.6	5805.4	5965.9	11327.3	7501.0
16	10000	1583.6	1565.0	1611.1	1617.0	3406.0	1643.7
64	40	326787.3	260593.3	520342.3	366766.2	319932.6	705985.9
64	300	137591.9	124444.1	167797.3	150135.6	206112.0	180832.2
64	2700	22213.7	21770.5	22823.4	23067.0	46544.6	23074.4
64	10000	6020.1	6427.2	6155.5	6514.5	13803.0	6194.7

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

Dimension: Overhead attribution

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

status: ok

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

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

scheme	concurrency	tier (microseconds)	submit side	handoff	body	return	status
apscheduler	1	40	107.12 microseconds	80.51 microseconds	57.45 microseconds	41.85 microseconds	ok
apscheduler	4	40	89.24 microseconds	511.93 microseconds	58.52 microseconds	235.26 microseconds	ok
apscheduler	1	10000	180.64 microseconds	129.18 microseconds	9.644 milliseconds	63.22 microseconds	ok
apscheduler	4	10000	128.64 microseconds	10.573 milliseconds	14.891 milliseconds	9.882 milliseconds	ok
asyncio_pool	1	40	17.11 microseconds	25.55 microseconds	47.21 microseconds	58.41 microseconds	ok
asyncio_pool	4	40	17.90 microseconds	290.31 microseconds	55.58 microseconds	275.71 microseconds	ok
asyncio_pool	1	10000	22.76 microseconds	35.65 microseconds	9.380 milliseconds	84.75 microseconds	ok
asyncio_pool	4	10000	24.43 microseconds	12.321 milliseconds	14.391 milliseconds	9.524 milliseconds	ok
bare_thread	1	40	142.34 microseconds	-75.72 microseconds	52.62 microseconds	30.13 microseconds	ok
bare_thread	4	40	149.38 microseconds	-73.41 microseconds	50.50 microseconds	299.38 microseconds	ok
bare_thread	1	10000	5.174 milliseconds	-5094.15 microseconds	9.410 milliseconds	37.10 microseconds	ok
bare_thread	4	10000	5.181 milliseconds	-5083.47 microseconds	14.567 milliseconds	9.130 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	8.07 microseconds	10.50 microseconds	44.59 microseconds	15.18 microseconds	ok
thread_pool	4	40	11.09 microseconds	187.33 microseconds	50.57 microseconds	155.11 microseconds	ok
thread_pool	1	10000	11.95 microseconds	16.89 microseconds	9.546 milliseconds	27.48 microseconds	ok
thread_pool	4	10000	13.35 microseconds	10.158 milliseconds	18.653 milliseconds	9.036 milliseconds	ok
zoo	1	40	51.07 microseconds	25.19 microseconds	52.73 microseconds	64.23 microseconds	ok
zoo	4	40	66.95 microseconds	358.54 microseconds	55.25 microseconds	319.43 microseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	87.78 microseconds	31.70 microseconds	9.652 milliseconds	97.67 microseconds	ok
zoo	4	10000	82.24 microseconds	10.273 milliseconds	14.148 milliseconds	9.899 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	-99.89 microseconds	return	22.38 microseconds
asyncio_pool	1	40	42.90 microseconds	submit side	33.97 microseconds
bare_thread	1	40	43.72 microseconds	handoff	100.91 microseconds
thread_pool	1	40	114.47 microseconds	return	49.05 microseconds
apscheduler	4	40	-138.41 microseconds	return	84.17 microseconds
asyncio_pool	4	40	78.10 microseconds	handoff	68.23 microseconds
bare_thread	4	40	342.78 microseconds	handoff	431.96 microseconds
thread_pool	4	40	373.11 microseconds	handoff	171.21 microseconds
apscheduler	1	10000	-151.45 microseconds	return	34.45 microseconds
asyncio_pool	1	10000	336.94 microseconds	body	271.67 microseconds
bare_thread	1	10000	341.61 microseconds	handoff	5.126 milliseconds
thread_pool	1	10000	264.05 microseconds	body	106.11 microseconds
apscheduler	4	10000	-764.89 microseconds	return	16.48 microseconds
asyncio_pool	4	10000	434.00 microseconds	return	374.94 microseconds
bare_thread	4	10000	9.711 milliseconds	handoff	15.357 milliseconds
thread_pool	4	10000	671.38 microseconds	return	862.47 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 190.97 microseconds vs with 193.02 microseconds, a difference of 2.05 microseconds; zoo 40 microseconds /

concurrency 4: without instrumentation 749.61 microseconds vs with 769.88 microseconds, a difference of 20.27 microseconds; zoo 10000 microseconds / concurrency 1: without instrumentation 9.779 milliseconds vs with 9.868 milliseconds, a difference of 89.14 microseconds; zoo 10000 microseconds / concurrency 4: without instrumentation 39.848 milliseconds vs with 39.597 milliseconds, a difference of 251.34 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 55.7%; zoo 10000 microseconds / concurrency 1 overlaps 16.8%; zoo 10000 microseconds / concurrency 4 overlaps 97.9%

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) 2.16 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 24.60 microseconds; per-round policy lookups (period / phase / timeout each read config) 17.00 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 7.30 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) 37.56 microseconds; per-round policy lookups (period / phase / timeout each read config) 30.47 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 5.07 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 3.41 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 41.16 microseconds; per-round policy lookups (period / phase / timeout each read config) 29.77 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 12.94 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) 46.12 microseconds; per-round policy lookups (period / phase / timeout each read config) 34.15 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 7.59 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test is not slower overall than `apscheduler` (-100 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 43 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 -0 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 44 microseconds more per task than `bare_thread`, most of it on 'handoff' (+101 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -91 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 114 microseconds more per task than `thread_pool`, most of it on 'return' (+49 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` (-138 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 78 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+68 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -0 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 343 microseconds more per task than `bare_thread`, most of it on 'handoff' (+432 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -82 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 373 microseconds more per task than `thread_pool`, most of it on 'handoff' (+171 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) 38, per-round policy lookups (period / phase / timeout each read config) 30, submit side, other (the adapter's own bookkeeping and completion-signal state) 5.
- At concurrency 1 and a 10000 microsecond body, the framework under test is not slower overall than `apscheduler` (-151 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 337 microseconds more per task than `asyncio_pool`, most of it on 'body' (+272 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -4 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 342 microseconds more per task than `bare_thread`, most of it on 'handoff' (+5126 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5087 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 264 microseconds more per task than `thread_pool`, most of it on 'body' (+106 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 is not slower overall than `apscheduler` (-765 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 434 microseconds more per task than `asyncio_pool`, most of it on 'return' (+375 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -2047 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 9711 microseconds more per task than `bare_thread`, most of it on 'handoff' (+15357 microseconds).

- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5098 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 671 microseconds more per task than thread_pool, most of it on 'return' (+862 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'body' is -4505 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) 46, per-round policy lookups (period / phase / timeout each read config) 34, submit side, other (the adapter's own bookkeeping and completion-signal state) 8.

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 1.62x faster than apscheduler	174.06 microseconds
process_pool	1	40	the framework under test is 2.12x faster than process_pool	174.06 microseconds
apscheduler	1	300	the framework under test is 1.26x faster than apscheduler	433.22 microseconds
process_pool	1	300	the framework under test is 1.42x faster than process_pool	433.22 microseconds
apscheduler	1	2700	the framework under test is 1.03x faster than apscheduler	2.800 milliseconds
process_pool	1	2700	the framework under test is 1.07x faster than process_pool	2.800 milliseconds
apscheduler	1	10000	the framework under test is 1.03x faster than apscheduler	9.885 milliseconds
process_pool	1	10000	the framework under test is 1.02x faster than process_pool	9.885 milliseconds
apscheduler	4	40	the framework under test is 1.28x faster than apscheduler	200.10 microseconds
apscheduler	4	300	the framework under test is 1.13x faster than apscheduler	457.43 microseconds
apscheduler	4	2700	the framework under test is 1.06x faster than apscheduler	2.814 milliseconds
apscheduler	16	40	the framework under test is 1.23x faster than apscheduler	199.72 microseconds
apscheduler	16	300	the framework under test is 1.12x faster than apscheduler	464.10 microseconds
apscheduler	16	2700	the framework under test is 1.04x faster than apscheduler	2.869 milliseconds
apscheduler	64	40	the framework under test is 1.25x faster than apscheduler	195.85 microseconds
apscheduler	64	300	the framework under test is 1.11x faster than apscheduler	465.14 microseconds

Openly published unfavourable data

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

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
thread_pool	1	40	thread_pool is 2.25x faster than zoo	174.06 microseconds
thread_pool	1	300	thread_pool is 1.34x faster than zoo	433.22 microseconds
asyncio_pool	1	2700	asyncio_pool is 1.04x faster than zoo	2.800 milliseconds
bare_thread	1	2700	bare_thread is 1.04x faster than zoo	2.800 milliseconds
thread_pool	1	2700	thread_pool is 1.05x faster than zoo	2.800 milliseconds
asyncio_pool	1	10000	asyncio_pool is 1.02x faster than zoo	9.885 milliseconds
bare_thread	1	10000	bare_thread is 1.04x faster than zoo	9.885 milliseconds
thread_pool	1	10000	thread_pool is 1.02x faster than zoo	9.885 milliseconds
asyncio_pool	4	40	asyncio_pool is 1.28x faster than zoo	200.10 microseconds
thread_pool	4	40	thread_pool is 2.08x faster than zoo	200.10 microseconds
asyncio_pool	4	300	asyncio_pool is 1.12x faster than zoo	457.43 microseconds
bare_thread	4	300	bare_thread is 1.11x faster than zoo	457.43 microseconds
process_pool	4	300	process_pool is 1.34x faster than zoo	457.43 microseconds
thread_pool	4	300	thread_pool is 1.30x faster than zoo	457.43 microseconds
bare_thread	4	2700	bare_thread is 1.03x faster than zoo	2.814 milliseconds
process_pool	4	2700	process_pool is 2.07x faster than zoo	2.814 milliseconds
thread_pool	4	2700	thread_pool is 1.04x faster than zoo	2.814 milliseconds
asyncio_pool	4	10000	asyncio_pool is 1.03x faster than zoo	10.050 milliseconds
bare_thread	4	10000	bare_thread is 1.02x faster than zoo	10.050 milliseconds
process_pool	4	10000	process_pool is 2.16x faster than zoo	10.050 milliseconds
thread_pool	4	10000	thread_pool is 1.02x faster than zoo	10.050 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
asyncio_pool	16	40	asyncio_pool is 1.53x faster than zoo	199.72 microseconds
bare_thread	16	40	bare_thread is 1.17x faster than zoo	199.72 microseconds
process_pool	16	40	process_pool is 1.31x faster than zoo	199.72 microseconds
thread_pool	16	40	thread_pool is 2.24x faster than zoo	199.72 microseconds
asyncio_pool	16	300	asyncio_pool is 1.18x faster than zoo	464.10 microseconds
bare_thread	16	300	bare_thread is 1.11x faster than zoo	464.10 microseconds
process_pool	16	300	process_pool is 1.67x faster than zoo	464.10 microseconds
thread_pool	16	300	thread_pool is 1.31x faster than zoo	464.10 microseconds
asyncio_pool	16	2700	asyncio_pool is 1.04x faster than zoo	2.869 milliseconds
bare_thread	16	2700	bare_thread is 1.07x faster than zoo	2.869 milliseconds
process_pool	16	2700	process_pool is 2.03x faster than zoo	2.869 milliseconds
thread_pool	16	2700	thread_pool is 1.35x faster than zoo	2.869 milliseconds
asyncio_pool	16	10000	asyncio_pool is 1.02x faster than zoo	10.104 milliseconds
bare_thread	16	10000	bare_thread is 1.02x faster than zoo	10.104 milliseconds
process_pool	16	10000	process_pool is 2.15x faster than zoo	10.104 milliseconds
thread_pool	16	10000	thread_pool is 1.04x faster than zoo	10.104 milliseconds
asyncio_pool	64	40	asyncio_pool is 1.59x faster than zoo	195.85 microseconds
bare_thread	64	40	bare_thread is 1.12x faster than zoo	195.85 microseconds
thread_pool	64	40	thread_pool is 2.16x faster than zoo	195.85 microseconds
asyncio_pool	64	300	asyncio_pool is 1.22x faster than zoo	465.14 microseconds
bare_thread	64	300	bare_thread is 1.09x faster than zoo	465.14 microseconds
process_pool	64	300	process_pool is 1.50x faster than zoo	465.14 microseconds
thread_pool	64	300	thread_pool is 1.31x faster than zoo	465.14 microseconds
asyncio_pool	64	2700	asyncio_pool is 1.03x faster than zoo	2.881 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
bare_thread	64	2700	bare_thread is 1.04x faster than zoo	2.881 milliseconds
process_pool	64	2700	process_pool is 2.10x faster than zoo	2.881 milliseconds
thread_pool	64	2700	thread_pool is 1.04x faster than zoo	2.881 milliseconds
apscheduler	64	10000	apscheduler is 1.07x faster than zoo	10.631 milliseconds
bare_thread	64	10000	bare_thread is 1.08x faster than zoo	10.631 milliseconds
process_pool	64	10000	process_pool is 2.29x faster than zoo	10.631 milliseconds

tiers with no winner

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

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

band this run: 1.0% to 35.2% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	1	40	1.23x	the two sides differ by 22.6%, less than the band of 35.2% - no winner
bare_thread	1	40	0.83x	the two sides differ by 16.8%, less than the band of 19.7% - no winner
asyncio_pool	1	300	0.94x	the two sides differ by 6.4%, less than the band of 8.8% - no winner
bare_thread	1	300	0.92x	the two sides differ by 8.4%, less than the band of 9.5% - no winner
bare_thread	4	40	0.84x	the two sides differ by 15.7%, less than the band of 27.2% - no winner
process_pool	4	40	0.96x	the two sides differ by 4.4%, less than the band of 16.2% - no winner

baseline	concurrency	body tier (microseconds)	ratio	reason
asyncio_pool	4	2700	0.98x	the two sides differ by 1.8%, less than the band of 3.1% - no winner
apscheduler	4	10000	1.01x	the two sides differ by 0.5%, less than the band of 1.4% - no winner
apscheduler	16	10000	1.01x	the two sides differ by 1.2%, less than the band of 1.7% - no winner
process_pool	64	40	1.02x	the two sides differ by 2.1%, less than the band of 11.5% - no winner
apscheduler	64	2700	1.02x	the two sides differ by 2.0%, less than the band of 3.0% - no winner
asyncio_pool	64	10000	0.98x	the two sides differ by 2.2%, less than the band of 3.8% - no winner
thread_pool	64	10000	0.97x	the two sides differ by 2.8%, less than the band of 3.4% - 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': 29954, 'child_pids': [30410, 30413, 30416, 30419, 30422, 30428, 30434, 30440, 30446, 30464, 30482, 30500, 30521, 30577, 30634, 30692, 30758, 30760, 30762, 30764, 30766, 30771, 30776, 30781, 30786, 30803, 30821, 30838, 30856, 30918, 30977, 31039, 31104, 31138, 31172, 31206, 31240, 31373, 31506, 31639, 31772, 32301, 32830, 33359, 33888, 36001, 38115, 40228, 42343, 42347, 42351, 42355, 42359, 42366, 42373, 42380, 42387, 42406, 42425, 42444, 42463, 42530, 42597, 42664, 42732, 42734, 42736, 42738, 42740, 42745, 42750, 42755, 42760, 42777, 42794, 42811, 42829,

42885, 42941, 43000, 43068, 43070, 43072, 43074, 43076, 43081, 43086, 43091, 43096, 43113, 43130, 43147, 43164, 43205, 43239, 43280], '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	281.24 microseconds	330.91 microseconds	334.09 microseconds	0.071
apscheduler	1	300	544.52 microseconds	572.11 microseconds	578.66 microseconds	0.035
apscheduler	1	2700	2.884 milliseconds	2.955 milliseconds	3.076 milliseconds	0.019
apscheduler	1	10000	10.156 milliseconds	10.294 milliseconds	10.376 milliseconds	0.013
apscheduler	4	40	255.42 microseconds	282.19 microseconds	297.32 microseconds	0.047
apscheduler	4	300	516.26 microseconds	547.34 microseconds	563.64 microseconds	0.029
apscheduler	4	2700	2.983 milliseconds	3.037 milliseconds	3.077 milliseconds	0.013
apscheduler	4	10000	10.101 milliseconds	10.215 milliseconds	10.306 milliseconds	0.008
apscheduler	16	40	246.53 microseconds	260.75 microseconds	274.95 microseconds	0.043
apscheduler	16	300	520.81 microseconds	531.42 microseconds	533.12 microseconds	0.021
apscheduler	16	2700	2.998 milliseconds	3.061 milliseconds	3.068 milliseconds	0.015
apscheduler	16	10000	10.223 milliseconds	10.325 milliseconds	10.429 milliseconds	0.011
apscheduler	64	40	245.59 microseconds	261.27 microseconds	273.04 microseconds	0.058
apscheduler	64	300	514.29 microseconds	540.01 microseconds	549.92 microseconds	0.031
apscheduler	64	2700	2.940 milliseconds	3.031 milliseconds	3.147 milliseconds	0.020
apscheduler	64	10000	9.958 milliseconds	10.331 milliseconds	10.436 milliseconds	0.021
asyncio_pool	1	40	213.36 microseconds	294.34 microseconds	303.06 microseconds	0.254
asyncio_pool	1	300	405.28 microseconds	432.68 microseconds	436.27 microseconds	0.034
asyncio_pool	1	2700	2.701 milliseconds	2.740 milliseconds	2.742 milliseconds	0.007

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
asyncio_pool	1	10000	9.700 milliseconds	9.791 milliseconds	9.826 milliseconds	0.007
asyncio_pool	4	40	156.05 microseconds	178.57 microseconds	186.23 microseconds	0.081
asyncio_pool	4	300	407.59 microseconds	435.64 microseconds	443.37 microseconds	0.030
asyncio_pool	4	2700	2.764 milliseconds	2.847 milliseconds	2.873 milliseconds	0.016
asyncio_pool	4	10000	9.786 milliseconds	9.908 milliseconds	9.942 milliseconds	0.006
asyncio_pool	16	40	130.72 microseconds	144.29 microseconds	146.15 microseconds	0.048
asyncio_pool	16	300	392.41 microseconds	411.25 microseconds	422.56 microseconds	0.024
asyncio_pool	16	2700	2.756 milliseconds	2.816 milliseconds	2.822 milliseconds	0.012
asyncio_pool	16	10000	9.931 milliseconds	10.004 milliseconds	10.010 milliseconds	0.007
asyncio_pool	64	40	123.00 microseconds	126.31 microseconds	129.42 microseconds	0.022
asyncio_pool	64	300	381.41 microseconds	391.30 microseconds	417.56 microseconds	0.024
asyncio_pool	64	2700	2.804 milliseconds	2.846 milliseconds	2.857 milliseconds	0.008
asyncio_pool	64	10000	10.397 milliseconds	10.561 milliseconds	10.584 milliseconds	0.018
bare_thread	1	40	144.76 microseconds	175.84 microseconds	198.18 microseconds	0.099
bare_thread	1	300	396.70 microseconds	436.06 microseconds	444.03 microseconds	0.042
bare_thread	1	2700	2.680 milliseconds	2.726 milliseconds	2.739 milliseconds	0.009
bare_thread	1	10000	9.548 milliseconds	9.655 milliseconds	9.731 milliseconds	0.007
bare_thread	4	40	168.63 microseconds	252.60 microseconds	257.71 microseconds	0.215
bare_thread	4	300	412.07 microseconds	434.78 microseconds	440.21 microseconds	0.026
bare_thread	4	2700	2.728 milliseconds	2.756 milliseconds	2.765 milliseconds	0.007
bare_thread	4	10000	9.829 milliseconds	9.887 milliseconds	9.924 milliseconds	0.004
bare_thread	16	40	170.14 microseconds	176.65 microseconds	177.51 microseconds	0.025
bare_thread	16	300	419.21 microseconds	428.16 microseconds	436.10 microseconds	0.013
bare_thread	16	2700	2.682 milliseconds	2.699 milliseconds	2.700 milliseconds	0.004

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
bare_thread	16	10000	9.895 milliseconds	9.953 milliseconds	10.039 milliseconds	0.004
bare_thread	64	40	174.50 microseconds	206.96 microseconds	208.44 microseconds	0.082
bare_thread	64	300	426.28 microseconds	434.04 microseconds	440.99 microseconds	0.010
bare_thread	64	2700	2.775 milliseconds	2.925 milliseconds	2.984 milliseconds	0.021
bare_thread	64	10000	9.824 milliseconds	9.862 milliseconds	10.052 milliseconds	0.006
process_pool	1	40	369.75 microseconds	523.35 microseconds	702.27 microseconds	0.221
process_pool	1	300	614.59 microseconds	646.64 microseconds	657.87 microseconds	0.031
process_pool	1	2700	2.982 milliseconds	3.078 milliseconds	3.162 milliseconds	0.016
process_pool	1	10000	10.087 milliseconds	10.228 milliseconds	10.606 milliseconds	0.013
process_pool	4	40	191.30 microseconds	226.31 microseconds	233.00 microseconds	0.105
process_pool	4	300	341.25 microseconds	392.67 microseconds	397.76 microseconds	0.111
process_pool	4	2700	1.359 milliseconds	1.636 milliseconds	2.046 milliseconds	0.120
process_pool	4	10000	4.650 milliseconds	5.143 milliseconds	5.672 milliseconds	0.055
process_pool	16	40	152.53 microseconds	195.85 microseconds	222.74 microseconds	0.159
process_pool	16	300	277.10 microseconds	325.51 microseconds	342.76 microseconds	0.089
process_pool	16	2700	1.413 milliseconds	1.522 milliseconds	1.531 milliseconds	0.035
process_pool	16	10000	4.698 milliseconds	4.896 milliseconds	4.946 milliseconds	0.019
process_pool	64	40	200.04 microseconds	241.82 microseconds	253.06 microseconds	0.095
process_pool	64	300	310.51 microseconds	345.27 microseconds	355.80 microseconds	0.059
process_pool	64	2700	1.375 milliseconds	1.448 milliseconds	1.489 milliseconds	0.023
process_pool	64	10000	4.637 milliseconds	4.695 milliseconds	4.742 milliseconds	0.007
thread_pool	1	40	77.28 microseconds	97.73 microseconds	131.24 microseconds	0.178
thread_pool	1	300	324.43 microseconds	343.06 microseconds	365.55 microseconds	0.038
thread_pool	1	2700	2.658 milliseconds	2.708 milliseconds	2.729 milliseconds	0.009

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
thread_pool	1	10000	9.698 milliseconds	9.755 milliseconds	9.835 milliseconds	0.006
thread_pool	4	40	96.24 microseconds	109.34 microseconds	114.24 microseconds	0.064
thread_pool	4	300	352.35 microseconds	369.30 microseconds	376.29 microseconds	0.021
thread_pool	4	2700	2.711 milliseconds	2.755 milliseconds	2.771 milliseconds	0.011
thread_pool	4	10000	9.811 milliseconds	9.987 milliseconds	10.010 milliseconds	0.008
thread_pool	16	40	88.97 microseconds	96.29 microseconds	96.75 microseconds	0.062
thread_pool	16	300	354.95 microseconds	365.09 microseconds	366.11 microseconds	0.013
thread_pool	16	2700	2.133 milliseconds	2.172 milliseconds	2.176 milliseconds	0.012
thread_pool	16	10000	9.734 milliseconds	9.914 milliseconds	10.042 milliseconds	0.009
thread_pool	64	40	90.65 microseconds	93.87 microseconds	102.53 microseconds	0.035
thread_pool	64	300	353.92 microseconds	414.87 microseconds	472.73 microseconds	0.084
thread_pool	64	2700	2.774 milliseconds	2.889 milliseconds	2.977 milliseconds	0.019
thread_pool	64	10000	10.331 milliseconds	10.440 milliseconds	10.460 milliseconds	0.014
zoo	1	40	174.06 microseconds	211.81 microseconds	226.31 microseconds	0.098
zoo	1	300	433.22 microseconds	475.65 microseconds	481.58 microseconds	0.054
zoo	1	2700	2.800 milliseconds	2.853 milliseconds	2.875 milliseconds	0.009
zoo	1	10000	9.885 milliseconds	9.996 milliseconds	10.110 milliseconds	0.007
zoo	4	40	200.10 microseconds	217.10 microseconds	218.93 microseconds	0.057
zoo	4	300	457.43 microseconds	477.75 microseconds	504.06 microseconds	0.031
zoo	4	2700	2.814 milliseconds	2.893 milliseconds	2.915 milliseconds	0.016
zoo	4	10000	10.050 milliseconds	10.125 milliseconds	10.177 milliseconds	0.006
zoo	16	40	199.72 microseconds	210.55 microseconds	214.54 microseconds	0.026
zoo	16	300	464.10 microseconds	477.83 microseconds	481.00 microseconds	0.016
zoo	16	2700	2.869 milliseconds	2.904 milliseconds	2.919 milliseconds	0.010

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
zoo	16	10000	10.104 milliseconds	10.191 milliseconds	10.304 milliseconds	0.006
zoo	64	40	195.85 microseconds	203.19 microseconds	204.54 microseconds	0.020
zoo	64	300	465.14 microseconds	470.93 microseconds	474.34 microseconds	0.009
zoo	64	2700	2.881 milliseconds	2.939 milliseconds	2.962 milliseconds	0.010
zoo	64	10000	10.631 milliseconds	10.833 milliseconds	11.023 milliseconds	0.020

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	232.64 microseconds	82.72%	48.60 microseconds
apscheduler	1	300	250.38 microseconds	45.98%	294.14 microseconds
apscheduler	1	2700	295.90 microseconds	10.26%	2.588 milliseconds
apscheduler	1	10000	365.37 microseconds	3.60%	9.790 milliseconds
apscheduler	4	40	199.69 microseconds	78.18%	55.73 microseconds
apscheduler	4	300	207.65 microseconds	40.22%	308.61 microseconds
apscheduler	4	2700	291.97 microseconds	9.79%	2.691 milliseconds
apscheduler	4	10000	-	-	14.850 milliseconds
apscheduler	16	40	-	-	52.28 microseconds
apscheduler	16	300	-	-	309.42 microseconds
apscheduler	16	2700	-	-	2.739 milliseconds
apscheduler	16	10000	-	-	15.698 milliseconds
apscheduler	64	40	-	-	50.46 microseconds
apscheduler	64	300	-	-	306.59 microseconds
apscheduler	64	2700	-	-	2.676 milliseconds
apscheduler	64	10000	-	-	15.208 milliseconds
asyncio_pool	1	40	148.75 microseconds	69.72%	64.61 microseconds
asyncio_pool	1	300	107.53 microseconds	26.53%	297.76 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
asyncio_pool	1	2700	123.69 microseconds	4.58%	2.578 milliseconds
asyncio_pool	1	10000	153.03 microseconds	1.58%	9.547 milliseconds
asyncio_pool	4	40	102.43 microseconds	65.64%	53.62 microseconds
asyncio_pool	4	300	104.85 microseconds	25.72%	302.74 microseconds
asyncio_pool	4	2700	111.01 microseconds	4.02%	2.652 milliseconds
asyncio_pool	4	10000	-	-	13.992 milliseconds
asyncio_pool	16	40	-	-	53.62 microseconds
asyncio_pool	16	300	-	-	304.98 microseconds
asyncio_pool	16	2700	-	-	2.632 milliseconds
asyncio_pool	16	10000	-	-	19.975 milliseconds
asyncio_pool	64	40	-	-	52.29 microseconds
asyncio_pool	64	300	-	-	302.33 microseconds
asyncio_pool	64	2700	-	-	2.675 milliseconds
asyncio_pool	64	10000	-	-	81.526 milliseconds
bare_thread	1	40	92.45 microseconds	63.87%	52.31 microseconds
bare_thread	1	300	101.31 microseconds	25.54%	295.39 microseconds
bare_thread	1	2700	114.23 microseconds	4.26%	2.566 milliseconds
bare_thread	1	10000	126.52 microseconds	1.33%	9.422 milliseconds
bare_thread	4	40	116.46 microseconds	69.06%	52.17 microseconds
bare_thread	4	300	118.13 microseconds	28.67%	293.94 microseconds
bare_thread	4	2700	127.88 microseconds	4.69%	2.601 milliseconds
bare_thread	4	10000	-	-	14.814 milliseconds
bare_thread	16	40	-	-	51.40 microseconds
bare_thread	16	300	-	-	286.58 microseconds
bare_thread	16	2700	-	-	2.539 milliseconds
bare_thread	16	10000	-	-	14.979 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
bare_thread	64	40	-	-	51.89 microseconds
bare_thread	64	300	-	-	296.66 microseconds
bare_thread	64	2700	-	-	2.640 milliseconds
bare_thread	64	10000	-	-	15.142 milliseconds
process_pool	1	40	313.46 microseconds	84.78%	56.29 microseconds
process_pool	1	300	304.04 microseconds	49.47%	310.54 microseconds
process_pool	1	2700	338.24 microseconds	11.34%	2.644 milliseconds
process_pool	1	10000	342.42 microseconds	3.39%	9.744 milliseconds
process_pool	4	40	101.51 microseconds	53.06%	89.79 microseconds
process_pool	4	300	-	-	568.22 microseconds
process_pool	4	2700	-	-	4.781 milliseconds
process_pool	4	10000	-	-	17.953 milliseconds
process_pool	16	40	-	-	91.18 microseconds
process_pool	16	300	-	-	583.81 microseconds
process_pool	16	2700	-	-	9.970 milliseconds
process_pool	16	10000	-	-	52.459 milliseconds
process_pool	64	40	-	-	91.01 microseconds
process_pool	64	300	-	-	570.05 microseconds
process_pool	64	2700	-	-	28.234 milliseconds
process_pool	64	10000	-	-	159.612 milliseconds
thread_pool	1	40	30.24 microseconds	39.13%	47.04 microseconds
thread_pool	1	300	36.08 microseconds	11.12%	288.35 microseconds
thread_pool	1	2700	48.20 microseconds	1.81%	2.610 milliseconds
thread_pool	1	10000	70.99 microseconds	0.73%	9.627 milliseconds
thread_pool	4	40	46.57 microseconds	48.39%	49.67 microseconds
thread_pool	4	300	53.93 microseconds	15.31%	298.42 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
thread_pool	4	2700	99.41 microseconds	3.67%	2.611 milliseconds
thread_pool	4	10000	-	-	15.069 milliseconds
thread_pool	16	40	-	-	48.79 microseconds
thread_pool	16	300	-	-	305.06 microseconds
thread_pool	16	2700	-	-	2.035 milliseconds
thread_pool	16	10000	-	-	19.866 milliseconds
thread_pool	64	40	-	-	47.76 microseconds
thread_pool	64	300	-	-	297.37 microseconds
thread_pool	64	2700	-	-	2.692 milliseconds
thread_pool	64	10000	-	-	59.527 milliseconds
zoo	1	40	123.88 microseconds	71.17%	50.17 microseconds
zoo	1	300	131.17 microseconds	30.28%	302.05 microseconds
zoo	1	2700	136.04 microseconds	4.86%	2.664 milliseconds
zoo	1	10000	147.92 microseconds	1.50%	9.737 milliseconds
zoo	4	40	144.96 microseconds	72.44%	55.14 microseconds
zoo	4	300	151.15 microseconds	33.04%	306.27 microseconds
zoo	4	2700	200.65 microseconds	7.13%	2.613 milliseconds
zoo	4	10000	-	-	14.817 milliseconds
zoo	16	40	-	-	52.89 microseconds
zoo	16	300	-	-	305.22 microseconds
zoo	16	2700	-	-	2.658 milliseconds
zoo	16	10000	-	-	20.089 milliseconds
zoo	64	40	-	-	52.68 microseconds
zoo	64	300	-	-	303.54 microseconds
zoo	64	2700	-	-	2.679 milliseconds
zoo	64	10000	-	-	51.276 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3555.7
apscheduler	1	300	1836.5
apscheduler	1	2700	346.7
apscheduler	1	10000	98.5
apscheduler	4	40	15660.3
apscheduler	4	300	7748.0
apscheduler	4	2700	1340.8
apscheduler	4	10000	396.0
apscheduler	16	40	64899.7
apscheduler	16	300	30721.3
apscheduler	16	2700	5337.6
apscheduler	16	10000	1565.0
apscheduler	64	40	260593.3
apscheduler	64	300	124444.1
apscheduler	64	2700	21770.5
apscheduler	64	10000	6427.2
asyncio_pool	1	40	4686.8
asyncio_pool	1	300	2467.4
asyncio_pool	1	2700	370.2
asyncio_pool	1	10000	103.1
asyncio_pool	4	40	25632.9
asyncio_pool	4	300	9813.8
asyncio_pool	4	2700	1447.4
asyncio_pool	4	10000	408.8
asyncio_pool	16	40	122396.1
asyncio_pool	16	300	40773.2
asyncio_pool	16	2700	5805.4
asyncio_pool	16	10000	1611.1
asyncio_pool	64	40	520342.3
asyncio_pool	64	300	167797.3
asyncio_pool	64	2700	22823.4
asyncio_pool	64	10000	6155.5
bare_thread	1	40	6907.9
bare_thread	1	300	2520.8
bare_thread	1	2700	373.2
bare_thread	1	10000	104.7
bare_thread	4	40	23720.3
bare_thread	4	300	9707.1
bare_thread	4	2700	1466.1
bare_thread	4	10000	407.0
bare_thread	16	40	94038.7
bare_thread	16	300	38167.2
bare_thread	16	2700	5965.9

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
bare_thread	16	10000	1617.0
bare_thread	64	40	366766.2
bare_thread	64	300	150135.6
bare_thread	64	2700	23067.0
bare_thread	64	10000	6514.5
process_pool	1	40	2704.5
process_pool	1	300	1627.1
process_pool	1	2700	335.3
process_pool	1	10000	99.1
process_pool	4	40	20909.5
process_pool	4	300	11721.5
process_pool	4	2700	2944.3
process_pool	4	10000	860.2
process_pool	16	40	104898.1
process_pool	16	300	57740.1
process_pool	16	2700	11327.3
process_pool	16	10000	3406.0
process_pool	64	40	319932.6
process_pool	64	300	206112.0
process_pool	64	2700	46544.6
process_pool	64	10000	13803.0
thread_pool	1	40	12939.1
thread_pool	1	300	3082.3
thread_pool	1	2700	376.2
thread_pool	1	10000	103.1
thread_pool	4	40	41562.7
thread_pool	4	300	11352.2
thread_pool	4	2700	1475.5
thread_pool	4	10000	407.7
thread_pool	16	40	179841.5
thread_pool	16	300	45076.5
thread_pool	16	2700	7501.0
thread_pool	16	10000	1643.7
thread_pool	64	40	705985.9
thread_pool	64	300	180832.2
thread_pool	64	2700	23074.4
thread_pool	64	10000	6194.7
zoo	1	40	5745.3
zoo	1	300	2308.3
zoo	1	2700	357.2
zoo	1	10000	101.2
zoo	4	40	19989.9
zoo	4	300	8744.6
zoo	4	2700	1421.5

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
zoo	4	10000	398.0
zoo	16	40	80111.5
zoo	16	300	34475.3
zoo	16	2700	5576.0
zoo	16	10000	1583.6
zoo	64	40	326787.3
zoo	64	300	137591.9
zoo	64	2700	22213.7
zoo	64	10000	6020.1