

zoo-bench performance report

raw data: results/zoo-framework-0.6.0/20261011T091515Z.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)	0.6.0
framework under test (module __version__)	0.1.1-beta (footnote only, not the version criterion)
drive-surface generation of the framework under test	previous-generation drive surface (workers is a writable attribute; the completion signal comes from the framework's internal callback)
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
command to reproduce	zoo-bench run --framework zoo-framework==0.6.0

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 10000 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, within the measured tiers `apscheduler` 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 `apscheduler` 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 `apscheduler` 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 `apscheduler` 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 `asyncio_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers `asyncio_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers `asyncio_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers `asyncio_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, within the measured tiers `bare_thread` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers `bare_thread` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers `bare_thread` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers `bare_thread` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, 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 4, within the measured tiers `process_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers `process_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers `process_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 1, within the measured tiers `thread_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 4, within the measured tiers `thread_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 16, within the measured tiers `thread_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.
- At concurrency 64, within the measured tiers `thread_pool` is **always faster** than the framework under test - tiers like these are exactly the openly published unfavourable data of this report.

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

Overhead threshold crossings

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

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

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

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

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

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

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

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

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

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

scheme	concurrency	threshold	first tier at or below the threshold (microseconds)	measured tiers (microseconds)	note
zoo	1	15.00%	10000	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	-	not observed: within the measured tiers this baseline is always faster than the framework under test
apscheduler	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
apscheduler	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
apscheduler	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
asyncio_pool	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	1	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	4	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	16	-	not observed: within the measured tiers this baseline is always faster than the framework under test
bare_thread	64	-	not observed: within the measured tiers this baseline is always faster than the framework under test

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

Charts

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

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

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

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

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

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

Dimension: Latency percentiles and jitter

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

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	606.25 microseconds	293.74 microseconds	145.49 microseconds	143.75 microseconds	336.43 microseconds	74.84 microseconds
1	300	867.95 microseconds	559.04 microseconds	389.99 microseconds	399.35 microseconds	606.27 microseconds	338.26 microseconds
1	2700	3.263 milliseconds	2.989 milliseconds	2.617 milliseconds	2.690 milliseconds	3.015 milliseconds	2.669 milliseconds
1	10000	10.505 milliseconds	9.889 milliseconds	9.708 milliseconds	9.552 milliseconds	9.702 milliseconds	9.577 milliseconds
4	40	611.17 microseconds	267.20 microseconds	153.05 microseconds	158.85 microseconds	192.86 microseconds	95.11 microseconds
4	300	866.98 microseconds	526.01 microseconds	428.07 microseconds	415.41 microseconds	283.82 microseconds	361.48 microseconds
4	2700	3.307 milliseconds	3.003 milliseconds	2.800 milliseconds	2.693 milliseconds	1.390 milliseconds	2.676 milliseconds
4	10000	10.700 milliseconds	10.132 milliseconds	9.695 milliseconds	9.872 milliseconds	4.608 milliseconds	9.867 milliseconds
16	40	610.25 microseconds	226.27 microseconds	138.37 microseconds	171.89 microseconds	161.53 microseconds	92.31 microseconds
16	300	878.71 microseconds	484.41 microseconds	395.42 microseconds	423.77 microseconds	278.33 microseconds	353.51 microseconds
16	2700	3.306 milliseconds	2.977 milliseconds	2.744 milliseconds	2.704 milliseconds	1.410 milliseconds	2.790 milliseconds
16	10000	10.596 milliseconds	10.332 milliseconds	9.964 milliseconds	9.972 milliseconds	4.636 milliseconds	10.146 milliseconds
64	40	565.92 microseconds	223.95 microseconds	120.73 microseconds	170.05 microseconds	196.54 microseconds	89.40 microseconds
64	300	842.59 microseconds	507.24 microseconds	371.91 microseconds	425.03 microseconds	300.97 microseconds	355.32 microseconds
64	2700	3.323 milliseconds	2.949 milliseconds	2.832 milliseconds	2.764 milliseconds	1.354 milliseconds	2.788 milliseconds
64	10000	10.636 milliseconds	10.585 milliseconds	10.189 milliseconds	9.526 milliseconds	4.629 milliseconds	10.289 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	88.13%	80.50%	66.60%	64.52%	83.87%	40.99%
1	300	62.15%	46.08%	25.72%	26.40%	49.32%	12.38%
1	2700	17.04%	8.91%	4.87%	4.18%	11.36%	1.76%
1	10000	5.28%	3.41%	1.51%	1.31%	3.30%	0.64%
4	40	89.54%	79.29%	64.75%	67.82%	51.69%	48.45%

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
4	300	62.78%	39.72%	24.92%	29.04%	-	14.55%
4	2700	19.63%	10.01%	4.93%	4.60%	-	2.23%
4	10000	-	-	-	-	-	-
16	40	-	-	-	-	-	-
16	300	-	-	-	-	-	-
16	2700	-	-	-	-	-	-
16	10000	-	-	-	-	-	-
64	40	-	-	-	-	-	-
64	300	-	-	-	-	-	-
64	2700	-	-	-	-	-	-
64	10000	-	-	-	-	-	-

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

Dimension: Throughput and concurrency scaling

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

concurrency	body tier (microseconds)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	1649.5	3404.3	6873.2	6956.3	2972.4	13362.6
1	300	1152.1	1788.8	2564.2	2504.1	1649.4	2956.3
1	2700	306.5	334.5	382.1	371.7	331.7	374.6
1	10000	95.2	101.1	103.0	104.7	103.1	104.4
4	40	6544.9	14970.0	26136.1	25180.5	20740.7	42055.6
4	300	4613.7	7604.4	9344.3	9629.0	14093.4	11065.7
4	2700	1209.4	1331.9	1428.5	1485.5	2878.6	1494.9
4	10000	373.8	394.8	412.6	405.2	868.1	405.4
16	40	26218.8	70711.5	115629.2	93082.7	99052.0	173329.2
16	300	18208.4	33030.1	40462.9	37756.1	57486.6	45260.1
16	2700	4839.3	5375.2	5831.5	5917.6	11345.1	5735.0
16	10000	1510.0	1548.6	1605.9	1604.5	3451.0	1577.0
64	40	113090.9	285777.7	530094.2	376362.9	325635.1	715882.7
64	300	75956.6	126173.2	172085.1	150577.9	212645.4	180117.2
64	2700	19258.5	21702.3	22599.7	23154.0	47269.4	22956.6
64	10000	6017.5	6046.2	6281.4	6718.2	13826.1	6220.2

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 (microse conds)	submit side	handoff	body	return	status
apscheduler	1	40	107.44 microseconds	91.43 microseconds	50.14 microseconds	39.97 microseconds	ok
apscheduler	4	40	110.35 microseconds	524.42 microseconds	55.76 microseconds	229.78 microseconds	ok
apscheduler	1	10000	138.62 microseconds	110.21 microseconds	9.664 milliseconds	49.29 microseconds	ok
apscheduler	4	10000	120.24 microseconds	12.806 milliseconds	14.493 milliseconds	9.873 milliseconds	ok
asyncio_pool	1	40	17.47 microseconds	26.05 microseconds	47.28 microseconds	58.04 microseconds	ok
asyncio_pool	4	40	18.91 microseconds	250.49 microseconds	53.92 microseconds	283.87 microseconds	ok
asyncio_pool	1	10000	23.03 microseconds	35.74 microseconds	9.574 milliseconds	84.68 microseconds	ok
asyncio_pool	4	10000	22.74 microseconds	12.394 milliseconds	10.117 milliseconds	10.093 milliseconds	ok
bare_thread	1	40	143.44 microseconds	-75.55 microseconds	51.49 microseconds	30.78 microseconds	ok
bare_thread	4	40	152.07 microseconds	-74.15 microseconds	50.96 microseconds	304.52 microseconds	ok
bare_thread	1	10000	5.171 milliseconds	-5093.61 microseconds	9.397 milliseconds	37.25 microseconds	ok
bare_thread	4	10000	5.180 milliseconds	-5085.51 microseconds	14.704 milliseconds	9.875 milliseconds	ok
process_pool	1	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)

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	8.13 microseconds	10.33 microseconds	46.76 microseconds	15.15 microseconds	ok
thread_pool	4	40	11.19 microseconds	173.04 microseconds	49.03 microseconds	153.53 microseconds	ok
thread_pool	1	10000	12.12 microseconds	16.27 microseconds	9.498 milliseconds	27.49 microseconds	ok
thread_pool	4	10000	13.25 microseconds	9.992 milliseconds	14.876 milliseconds	8.754 milliseconds	ok
zoo	1	40	23.37 microseconds	366.57 microseconds	71.55 microseconds	131.62 microseconds	ok
zoo	4	40	16.35 microseconds	1.144 milliseconds	67.00 microseconds	1.218 milliseconds	ok

scheme	concurrency	tier (microse conds)	submit side	handoff	body	return	status
zoo	1	10000	36.26 microseconds	392.84 microseconds	9.674 milliseconds	145.56 microseconds	ok
zoo	4	10000	21.46 microseconds	10.784 milliseconds	14.929 milliseconds	11.213 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	324.22 microseconds	handoff	275.14 microseconds
asyncio_pool	1	40	467.68 microseconds	handoff	340.52 microseconds
bare_thread	1	40	465.54 microseconds	handoff	442.12 microseconds
thread_pool	1	40	536.55 microseconds	handoff	356.24 microseconds
apscheduler	4	40	1.548 milliseconds	return	987.97 microseconds
asyncio_pool	4	40	1.870 milliseconds	return	933.88 microseconds
bare_thread	4	40	2.048 milliseconds	handoff	1.219 milliseconds
thread_pool	4	40	2.092 milliseconds	return	1.064 milliseconds
apscheduler	1	10000	313.01 microseconds	handoff	282.63 microseconds
asyncio_pool	1	10000	568.72 microseconds	handoff	357.10 microseconds
bare_thread	1	10000	766.30 microseconds	handoff	5.486 milliseconds
thread_pool	1	10000	708.50 microseconds	handoff	376.57 microseconds
apscheduler	4	10000	2.600 milliseconds	return	1.340 milliseconds
asyncio_pool	4	10000	2.688 milliseconds	body	4.812 milliseconds
bare_thread	4	10000	10.806 milliseconds	handoff	15.869 milliseconds
thread_pool	4	10000	3.498 milliseconds	return	2.459 milliseconds

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

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

seams not measured this time (this generation of the framework under test has no such entry point, so the related sub-measurements are absent): WorkerDispatchCore.resolve_period / resolve_phase / resolve_run_timeout (per-round policy lookups)

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 624.17 microseconds vs with 616.59 microseconds, a difference of 7.58 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost); zoo 40 microseconds / concurrency 4: without instrumentation 2.464 milliseconds vs with 2.477 milliseconds, a difference of 13.43

microseconds; zoo 10000 microseconds / concurrency 1: without instrumentation 10.256 milliseconds vs with 10.283 milliseconds, a difference of 26.40 microseconds; zoo 10000 microseconds / concurrency 4: without instrumentation 42.327 milliseconds vs with 42.213 milliseconds, a difference of 114.21 microseconds (faster with instrumentation: this round's machine drift exceeded the instrumentation cost)

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) 3.82 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 13.62 microseconds; per-round policy lookups (period / phase / timeout each read config) 0.00 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 5.84 microseconds
- concurrency 4, tier 40 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 3.20 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 10.58 microseconds; per-round policy lookups (period / phase / timeout each read config) 0.00 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 4.34 microseconds
- concurrency 1, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 5.62 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 19.72 microseconds; per-round policy lookups (period / phase / timeout each read config) 0.00 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 9.41 microseconds
- concurrency 4, tier 10000 microseconds: rest of the scheduling round (due checks, locks, list upkeep) 4.23 microseconds; dispatch (handed to the scheduling model, incl. pool enqueue) 15.24 microseconds; per-round policy lookups (period / phase / timeout each read config) 0.00 microseconds; submit side, other (the adapter's own bookkeeping and completion-signal state) 5.66 microseconds

Attribution conclusions

- At concurrency 1 and a 40 microsecond body, the framework under test spends 324 microseconds more per task than `apscheduler`, most of it on 'handoff' (+275 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -84 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 468 microseconds more per task than `asyncio_pool`, most of it on 'handoff' (+341 microseconds).
- At concurrency 1 and a 40 microsecond body, the framework under test spends 466 microseconds more per task than `bare_thread`, most of it on 'handoff' (+442 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -120 microseconds, so read them together.
- At concurrency 1 and a 40 microsecond body, the framework under test spends 537 microseconds more per task than `thread_pool`, most of it on 'handoff' (+356 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 4, dispatch (handed to the scheduling model, incl. pool enqueue) 14, per-round policy lookups (period / phase / timeout each read config) 0, submit side, other (the adapter's own bookkeeping and completion-signal state) 6.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 1548 microseconds more per task than `apscheduler`, most of it on 'return' (+988 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -94 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 1870 microseconds more per task than `asyncio_pool`, most of it on 'return' (+934 microseconds).

- note: the per-segment differences in this group do not all share a sign - 'submit side' is -3 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 2048 microseconds more per task than bare_thread, most of it on 'handoff' (+1219 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -136 microseconds, so read them together.
- At concurrency 4 and a 40 microsecond body, the framework under test spends 2092 microseconds more per task than thread_pool, most of it on 'return' (+1064 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) 11, per-round policy lookups (period / phase / timeout each read config) 0, 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 313 microseconds more per task than apscheduler, most of it on 'handoff' (+283 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -102 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 569 microseconds more per task than asyncio_pool, most of it on 'handoff' (+357 microseconds).
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 766 microseconds more per task than bare_thread, most of it on 'handoff' (+5486 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5135 microseconds, so read them together.
- At concurrency 1 and a 10000 microsecond body, the framework under test spends 709 microseconds more per task than thread_pool, most of it on 'handoff' (+377 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 6, dispatch (handed to the scheduling model, incl. pool enqueue) 20, per-round policy lookups (period / phase / timeout each read config) 0, submit side, other (the adapter's own bookkeeping and completion-signal state) 9.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 2600 microseconds more per task than apscheduler, most of it on 'return' (+1340 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -2022 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 2688 microseconds more per task than asyncio_pool, most of it on 'body' (+4812 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'handoff' is -1610 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 10806 microseconds more per task than bare_thread, most of it on 'handoff' (+15869 microseconds).
- note: the per-segment differences in this group do not all share a sign - 'submit side' is -5158 microseconds, so read them together.
- At concurrency 4 and a 10000 microsecond body, the framework under test spends 3498 microseconds more per task than thread_pool, most of it on 'return' (+2459 microseconds).
- In this group the framework's submit side consists of (microseconds): rest of the scheduling round (due checks, locks, list upkeep) 4, dispatch (handed to the scheduling model, incl. pool enqueue) 15, per-round policy lookups (period / phase / timeout each read config) 0, submit side, other (the adapter's own bookkeeping

and completion-signal state) 6.

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: not_measured (the conclusion was reached on the 'previous' drive-surface generation installed during this measurement)

on the version under test, **none of the enumerated semantic items is measurable**: either its action is not implemented, or its layer cannot execute the reactor. Per design D7's escape hatch, such items are dropped instead of being replaced with an approximate status

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)	no	the judgment exists but its action is not implemented : this probe drives through scheduling rounds and the probe worker declares a loop, so the rounds did run - a missing reap can only mean the judgment itself has no action. Enabling it would just add one subtraction and measure nothing of the semantics' cost
priority	event layer	structural probe (checks whether the symbols required for execution exist)	no	executing reactors on the event layer requires a perform symbol: EventWorker._execute calls reactor.perform (workers/event_worker.py), while EventReactor only defines execute (reactor/event_reactor.py) - a matched reactor would raise AttributeError at once; EventPriorityCalculator exists (fifo/node/event_fifo_node.py) but is not wired into the drain loop - 'sort by priority' in workers/event_worker.py is only a comment

semantics item	layer	how probed	measurable	evidence / reason
retry	event layer	structural probe (checks whether the symbols required for execution exist)	no	executing reactors on the event layer requires a perform symbol: EventWorker._execute calls reactor.perform (workers/event_worker.py), while EventReactor only defines execute (reactor/event_reactor.py) - a matched reactor would raise AttributeError at once; retry_times defaults to 0, a clean off state, but its trigger is 'no reactor matched' - and the reactor path itself cannot execute

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.

no tier within the measured set has the framework under test ahead.

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
apscheduler	1	40	apscheduler is 2.06x faster than zoo	606.25 microseconds
asyncio_pool	1	40	asyncio_pool is 4.17x faster than zoo	606.25 microseconds
bare_thread	1	40	bare_thread is 4.22x faster than zoo	606.25 microseconds
process_pool	1	40	process_pool is 1.80x faster than zoo	606.25 microseconds
thread_pool	1	40	thread_pool is 8.10x faster than zoo	606.25 microseconds
apscheduler	1	300	apscheduler is 1.55x faster than zoo	867.95 microseconds
asyncio_pool	1	300	asyncio_pool is 2.23x faster than zoo	867.95 microseconds
bare_thread	1	300	bare_thread is 2.17x faster than zoo	867.95 microseconds
process_pool	1	300	process_pool is 1.43x faster than zoo	867.95 microseconds
thread_pool	1	300	thread_pool is 2.57x faster than zoo	867.95 microseconds
apscheduler	1	2700	apscheduler is 1.09x faster than zoo	3.263 milliseconds
asyncio_pool	1	2700	asyncio_pool is 1.25x faster than zoo	3.263 milliseconds
bare_thread	1	2700	bare_thread is 1.21x faster than zoo	3.263 milliseconds
thread_pool	1	2700	thread_pool is 1.22x faster than zoo	3.263 milliseconds
apscheduler	1	10000	apscheduler is 1.06x faster than zoo	10.505 milliseconds
asyncio_pool	1	10000	asyncio_pool is 1.08x faster than zoo	10.505 milliseconds
bare_thread	1	10000	bare_thread is 1.10x faster than zoo	10.505 milliseconds
process_pool	1	10000	process_pool is 1.08x faster than zoo	10.505 milliseconds
thread_pool	1	10000	thread_pool is 1.10x faster than zoo	10.505 milliseconds
apscheduler	4	40	apscheduler is 2.29x faster than zoo	611.17 microseconds
asyncio_pool	4	40	asyncio_pool is 3.99x faster than zoo	611.17 microseconds
bare_thread	4	40	bare_thread is 3.85x faster than zoo	611.17 microseconds
process_pool	4	40	process_pool is 3.17x faster than zoo	611.17 microseconds
thread_pool	4	40	thread_pool is 6.43x faster than zoo	611.17 microseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
apscheduler	4	300	apscheduler is 1.65x faster than zoo	866.98 microseconds
asyncio_pool	4	300	asyncio_pool is 2.03x faster than zoo	866.98 microseconds
bare_thread	4	300	bare_thread is 2.09x faster than zoo	866.98 microseconds
process_pool	4	300	process_pool is 3.05x faster than zoo	866.98 microseconds
thread_pool	4	300	thread_pool is 2.40x faster than zoo	866.98 microseconds
asyncio_pool	4	2700	asyncio_pool is 1.18x faster than zoo	3.307 milliseconds
bare_thread	4	2700	bare_thread is 1.23x faster than zoo	3.307 milliseconds
process_pool	4	2700	process_pool is 2.38x faster than zoo	3.307 milliseconds
thread_pool	4	2700	thread_pool is 1.24x faster than zoo	3.307 milliseconds
apscheduler	4	10000	apscheduler is 1.06x faster than zoo	10.700 milliseconds
asyncio_pool	4	10000	asyncio_pool is 1.10x faster than zoo	10.700 milliseconds
bare_thread	4	10000	bare_thread is 1.08x faster than zoo	10.700 milliseconds
process_pool	4	10000	process_pool is 2.32x faster than zoo	10.700 milliseconds
apscheduler	16	40	apscheduler is 2.70x faster than zoo	610.25 microseconds
asyncio_pool	16	40	asyncio_pool is 4.41x faster than zoo	610.25 microseconds
bare_thread	16	40	bare_thread is 3.55x faster than zoo	610.25 microseconds
process_pool	16	40	process_pool is 3.78x faster than zoo	610.25 microseconds
thread_pool	16	40	thread_pool is 6.61x faster than zoo	610.25 microseconds
apscheduler	16	300	apscheduler is 1.81x faster than zoo	878.71 microseconds
asyncio_pool	16	300	asyncio_pool is 2.22x faster than zoo	878.71 microseconds
bare_thread	16	300	bare_thread is 2.07x faster than zoo	878.71 microseconds
process_pool	16	300	process_pool is 3.16x faster than zoo	878.71 microseconds
thread_pool	16	300	thread_pool is 2.49x faster than zoo	878.71 microseconds
apscheduler	16	2700	apscheduler is 1.11x faster than zoo	3.306 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
asyncio_pool	16	2700	asyncio_pool is 1.21x faster than zoo	3.306 milliseconds
bare_thread	16	2700	bare_thread is 1.22x faster than zoo	3.306 milliseconds
process_pool	16	2700	process_pool is 2.34x faster than zoo	3.306 milliseconds
thread_pool	16	2700	thread_pool is 1.19x faster than zoo	3.306 milliseconds
asyncio_pool	16	10000	asyncio_pool is 1.06x faster than zoo	10.596 milliseconds
bare_thread	16	10000	bare_thread is 1.06x faster than zoo	10.596 milliseconds
process_pool	16	10000	process_pool is 2.29x faster than zoo	10.596 milliseconds
thread_pool	16	10000	thread_pool is 1.04x faster than zoo	10.596 milliseconds
apscheduler	64	40	apscheduler is 2.53x faster than zoo	565.92 microseconds
asyncio_pool	64	40	asyncio_pool is 4.69x faster than zoo	565.92 microseconds
bare_thread	64	40	bare_thread is 3.33x faster than zoo	565.92 microseconds
process_pool	64	40	process_pool is 2.88x faster than zoo	565.92 microseconds
thread_pool	64	40	thread_pool is 6.33x faster than zoo	565.92 microseconds
apscheduler	64	300	apscheduler is 1.66x faster than zoo	842.59 microseconds
asyncio_pool	64	300	asyncio_pool is 2.27x faster than zoo	842.59 microseconds
bare_thread	64	300	bare_thread is 1.98x faster than zoo	842.59 microseconds
process_pool	64	300	process_pool is 2.80x faster than zoo	842.59 microseconds
thread_pool	64	300	thread_pool is 2.37x faster than zoo	842.59 microseconds
apscheduler	64	2700	apscheduler is 1.13x faster than zoo	3.323 milliseconds
asyncio_pool	64	2700	asyncio_pool is 1.17x faster than zoo	3.323 milliseconds
bare_thread	64	2700	bare_thread is 1.20x faster than zoo	3.323 milliseconds
process_pool	64	2700	process_pool is 2.45x faster than zoo	3.323 milliseconds
thread_pool	64	2700	thread_pool is 1.19x faster than zoo	3.323 milliseconds
asyncio_pool	64	10000	asyncio_pool is 1.04x faster than zoo	10.636 milliseconds

baseline	concurrency	body tier (microseconds)	gap	median of the framework under test
bare_thread	64	10000	bare_thread is 1.12x faster than zoo	10.636 milliseconds
process_pool	64	10000	process_pool is 2.30x faster than zoo	10.636 milliseconds
thread_pool	64	10000	thread_pool is 1.03x faster than zoo	10.636 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: 1.6% to 41.9% (it varies with each tier's spread).

baseline	concurrency	body tier (microseconds)	ratio	reason
process_pool	1	2700	0.92x	the two sides differ by 7.6%, less than the band of 21.2% - no winner
apscheduler	4	2700	0.91x	the two sides differ by 9.2%, less than the band of 9.6% - no winner
thread_pool	4	10000	0.92x	the two sides differ by 7.8%, less than the band of 9.2% - no winner
apscheduler	16	10000	0.98x	the two sides differ by 2.5%, less than the band of 5.1% - no winner
apscheduler	64	10000	1.00x	the two sides differ by 0.5%, less than the band of 4.1% - 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': 2921, 'child_pids': [3381, 3384, 3387, 3391, 3394, 3400, 3406, 3412, 3418, 3436, 3454, 3472, 3490, 3546, 3603, 3661, 3729, 3731, 3733, 3735, 3737, 3742, 3747, 3752, 3758, 3775, 3792, 3809, 3828, 3887, 3947, 4010, 4079, 4113, 4147, 4181, 4215, 4348, 4482, 4615, 4748, 5277, 5806, 6335, 6865, 8979, 11092, 13205, 15321, 15325, 15329, 15333, 15337, 15344, 15351, 15358, 15365, 15385, 15404, 15423, 15442, 15509, 15576, 15643, 15712, 15714, 15716, 15718, 15720, 15725, 15730, 15735, 15744, 15761, 15778, 15796, 15813, 15869, 15926, 16009, 16084, 16086, 16088, 16090, 16092, 16097, 16102, 16107, 16112, 16129, 16146, 16163, 16182, 16247, 16312, 16378], '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	293.74 microseconds	325.53 microseconds	342.96 microseconds	0.056
apscheduler	1	300	559.04 microseconds	588.66 microseconds	590.43 microseconds	0.031
apscheduler	1	2700	2.989 milliseconds	3.092 milliseconds	3.173 milliseconds	0.021
apscheduler	1	10000	9.889 milliseconds	10.123 milliseconds	10.143 milliseconds	0.011
apscheduler	4	40	267.20 microseconds	277.92 microseconds	308.26 microseconds	0.044
apscheduler	4	300	526.01 microseconds	708.75 microseconds	823.48 microseconds	0.142
apscheduler	4	2700	3.003 milliseconds	3.546 milliseconds	3.733 milliseconds	0.066
apscheduler	4	10000	10.132 milliseconds	10.472 milliseconds	10.635 milliseconds	0.015
apscheduler	16	40	226.27 microseconds	262.75 microseconds	266.37 microseconds	0.061
apscheduler	16	300	484.41 microseconds	511.45 microseconds	538.63 microseconds	0.032
apscheduler	16	2700	2.977 milliseconds	3.027 milliseconds	3.029 milliseconds	0.012

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
apscheduler	16	10000	10.332 milliseconds	10.686 milliseconds	11.247 milliseconds	0.022
apscheduler	64	40	223.95 microseconds	265.82 microseconds	274.59 microseconds	0.074
apscheduler	64	300	507.24 microseconds	538.33 microseconds	538.99 microseconds	0.029
apscheduler	64	2700	2.949 milliseconds	2.989 milliseconds	3.029 milliseconds	0.011
apscheduler	64	10000	10.585 milliseconds	10.720 milliseconds	10.771 milliseconds	0.027
asyncio_pool	1	40	145.49 microseconds	177.59 microseconds	207.40 microseconds	0.114
asyncio_pool	1	300	389.99 microseconds	421.09 microseconds	427.24 microseconds	0.035
asyncio_pool	1	2700	2.617 milliseconds	2.638 milliseconds	2.662 milliseconds	0.006
asyncio_pool	1	10000	9.708 milliseconds	9.924 milliseconds	10.234 milliseconds	0.015
asyncio_pool	4	40	153.05 microseconds	172.10 microseconds	184.08 microseconds	0.097
asyncio_pool	4	300	428.07 microseconds	462.16 microseconds	472.08 microseconds	0.040
asyncio_pool	4	2700	2.800 milliseconds	2.932 milliseconds	2.945 milliseconds	0.020
asyncio_pool	4	10000	9.695 milliseconds	9.778 milliseconds	9.863 milliseconds	0.007
asyncio_pool	16	40	138.37 microseconds	147.94 microseconds	148.85 microseconds	0.054
asyncio_pool	16	300	395.42 microseconds	411.17 microseconds	413.02 microseconds	0.021
asyncio_pool	16	2700	2.744 milliseconds	2.800 milliseconds	2.828 milliseconds	0.012
asyncio_pool	16	10000	9.964 milliseconds	10.041 milliseconds	10.050 milliseconds	0.004
asyncio_pool	64	40	120.73 microseconds	126.95 microseconds	128.50 microseconds	0.023
asyncio_pool	64	300	371.91 microseconds	375.19 microseconds	376.71 microseconds	0.007
asyncio_pool	64	2700	2.832 milliseconds	2.882 milliseconds	2.895 milliseconds	0.010
asyncio_pool	64	10000	10.189 milliseconds	10.383 milliseconds	10.530 milliseconds	0.019
bare_thread	1	40	143.75 microseconds	165.27 microseconds	169.11 microseconds	0.067
bare_thread	1	300	399.35 microseconds	468.24 microseconds	483.16 microseconds	0.070
bare_thread	1	2700	2.690 milliseconds	2.770 milliseconds	2.841 milliseconds	0.017

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
bare_thread	1	10000	9.552 milliseconds	9.644 milliseconds	9.659 milliseconds	0.005
bare_thread	4	40	158.85 microseconds	178.14 microseconds	191.10 microseconds	0.060
bare_thread	4	300	415.41 microseconds	440.37 microseconds	445.94 microseconds	0.029
bare_thread	4	2700	2.693 milliseconds	2.741 milliseconds	2.756 milliseconds	0.008
bare_thread	4	10000	9.872 milliseconds	9.967 milliseconds	10.315 milliseconds	0.012
bare_thread	16	40	171.89 microseconds	174.70 microseconds	176.22 microseconds	0.016
bare_thread	16	300	423.77 microseconds	440.14 microseconds	444.99 microseconds	0.018
bare_thread	16	2700	2.704 milliseconds	2.732 milliseconds	2.740 milliseconds	0.005
bare_thread	16	10000	9.972 milliseconds	10.392 milliseconds	10.652 milliseconds	0.017
bare_thread	64	40	170.05 microseconds	175.16 microseconds	176.83 microseconds	0.016
bare_thread	64	300	425.03 microseconds	431.55 microseconds	432.96 microseconds	0.008
bare_thread	64	2700	2.764 milliseconds	2.788 milliseconds	2.808 milliseconds	0.006
bare_thread	64	10000	9.526 milliseconds	9.558 milliseconds	9.578 milliseconds	0.002
process_pool	1	40	336.43 microseconds	360.56 microseconds	368.37 microseconds	0.047
process_pool	1	300	606.27 microseconds	631.93 microseconds	655.66 microseconds	0.033
process_pool	1	2700	3.015 milliseconds	3.702 milliseconds	5.552 milliseconds	0.198
process_pool	1	10000	9.702 milliseconds	9.785 milliseconds	9.839 milliseconds	0.005
process_pool	4	40	192.86 microseconds	251.09 microseconds	278.11 microseconds	0.172
process_pool	4	300	283.82 microseconds	328.17 microseconds	371.71 microseconds	0.083
process_pool	4	2700	1.390 milliseconds	1.547 milliseconds	1.670 milliseconds	0.054
process_pool	4	10000	4.608 milliseconds	4.892 milliseconds	5.049 milliseconds	0.025
process_pool	16	40	161.53 microseconds	206.04 microseconds	231.93 microseconds	0.146
process_pool	16	300	278.33 microseconds	329.96 microseconds	357.83 microseconds	0.089
process_pool	16	2700	1.410 milliseconds	1.508 milliseconds	1.542 milliseconds	0.035

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
process_pool	16	10000	4.636 milliseconds	5.021 milliseconds	5.550 milliseconds	0.049
process_pool	64	40	196.54 microseconds	237.84 microseconds	243.91 microseconds	0.097
process_pool	64	300	300.97 microseconds	327.19 microseconds	328.16 microseconds	0.047
process_pool	64	2700	1.354 milliseconds	1.384 milliseconds	1.460 milliseconds	0.023
process_pool	64	10000	4.629 milliseconds	4.673 milliseconds	4.729 milliseconds	0.006
thread_pool	1	40	74.84 microseconds	101.46 microseconds	112.12 microseconds	0.149
thread_pool	1	300	338.26 microseconds	362.96 microseconds	376.81 microseconds	0.039
thread_pool	1	2700	2.669 milliseconds	2.698 milliseconds	2.734 milliseconds	0.007
thread_pool	1	10000	9.577 milliseconds	9.658 milliseconds	9.749 milliseconds	0.007
thread_pool	4	40	95.11 microseconds	103.23 microseconds	106.77 microseconds	0.053
thread_pool	4	300	361.48 microseconds	375.03 microseconds	376.09 microseconds	0.018
thread_pool	4	2700	2.676 milliseconds	2.763 milliseconds	2.777 milliseconds	0.016
thread_pool	4	10000	9.867 milliseconds	11.864 milliseconds	12.865 milliseconds	0.081
thread_pool	16	40	92.31 microseconds	97.34 microseconds	98.25 microseconds	0.039
thread_pool	16	300	353.51 microseconds	362.47 microseconds	375.59 microseconds	0.023
thread_pool	16	2700	2.790 milliseconds	2.832 milliseconds	2.868 milliseconds	0.010
thread_pool	16	10000	10.146 milliseconds	10.195 milliseconds	10.211 milliseconds	0.005
thread_pool	64	40	89.40 microseconds	91.59 microseconds	92.21 microseconds	0.019
thread_pool	64	300	355.32 microseconds	389.22 microseconds	457.84 microseconds	0.069
thread_pool	64	2700	2.788 milliseconds	2.843 milliseconds	2.858 milliseconds	0.012
thread_pool	64	10000	10.289 milliseconds	10.354 milliseconds	10.413 milliseconds	0.012
zoo	1	40	606.25 microseconds	961.39 microseconds	991.89 microseconds	0.270
zoo	1	300	867.95 microseconds	1.228 milliseconds	1.250 milliseconds	0.252
zoo	1	2700	3.263 milliseconds	3.328 milliseconds	3.388 milliseconds	0.014

scheme	concurrency	body tier (microseconds)	median	p95	p99	relative spread
zoo	1	10000	10.505 milliseconds	10.893 milliseconds	10.970 milliseconds	0.021
zoo	4	40	611.17 microseconds	713.60 microseconds	748.92 microseconds	0.115
zoo	4	300	866.98 microseconds	898.40 microseconds	902.02 microseconds	0.041
zoo	4	2700	3.307 milliseconds	3.447 milliseconds	3.485 milliseconds	0.030
zoo	4	10000	10.700 milliseconds	10.837 milliseconds	10.853 milliseconds	0.012
zoo	16	40	610.25 microseconds	709.55 microseconds	747.97 microseconds	0.118
zoo	16	300	878.71 microseconds	976.84 microseconds	1.017 milliseconds	0.081
zoo	16	2700	3.306 milliseconds	3.405 milliseconds	3.495 milliseconds	0.029
zoo	16	10000	10.596 milliseconds	10.693 milliseconds	11.836 milliseconds	0.030
zoo	64	40	565.92 microseconds	759.15 microseconds	837.97 microseconds	0.170
zoo	64	300	842.59 microseconds	1.054 milliseconds	1.138 milliseconds	0.128
zoo	64	2700	3.323 milliseconds	3.551 milliseconds	3.583 milliseconds	0.035
zoo	64	10000	10.636 milliseconds	11.013 milliseconds	11.146 milliseconds	0.014

Appendix: Framework overhead share

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	1	40	236.46 microseconds	80.50%	57.28 microseconds
apscheduler	1	300	257.58 microseconds	46.08%	301.46 microseconds
apscheduler	1	2700	266.41 microseconds	8.91%	2.723 milliseconds
apscheduler	1	10000	336.85 microseconds	3.41%	9.552 milliseconds
apscheduler	4	40	211.85 microseconds	79.29%	55.35 microseconds
apscheduler	4	300	208.91 microseconds	39.72%	317.10 microseconds
apscheduler	4	2700	300.58 microseconds	10.01%	2.703 milliseconds
apscheduler	4	10000	-	-	14.772 milliseconds
apscheduler	16	40	-	-	53.66 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
apscheduler	16	300	-	-	299.58 microseconds
apscheduler	16	2700	-	-	2.719 milliseconds
apscheduler	16	10000	-	-	21.563 milliseconds
apscheduler	64	40	-	-	50.45 microseconds
apscheduler	64	300	-	-	304.01 microseconds
apscheduler	64	2700	-	-	2.679 milliseconds
apscheduler	64	10000	-	-	30.342 milliseconds
asyncio_pool	1	40	96.90 microseconds	66.60%	48.59 microseconds
asyncio_pool	1	300	100.32 microseconds	25.72%	289.67 microseconds
asyncio_pool	1	2700	127.46 microseconds	4.87%	2.490 milliseconds
asyncio_pool	1	10000	146.90 microseconds	1.51%	9.561 milliseconds
asyncio_pool	4	40	99.09 microseconds	64.75%	53.96 microseconds
asyncio_pool	4	300	106.69 microseconds	24.92%	321.38 microseconds
asyncio_pool	4	2700	138.11 microseconds	4.93%	2.662 milliseconds
asyncio_pool	4	10000	-	-	9.919 milliseconds
asyncio_pool	16	40	-	-	53.59 microseconds
asyncio_pool	16	300	-	-	305.70 microseconds
asyncio_pool	16	2700	-	-	2.620 milliseconds
asyncio_pool	16	10000	-	-	24.461 milliseconds
asyncio_pool	64	40	-	-	52.03 microseconds
asyncio_pool	64	300	-	-	293.02 microseconds
asyncio_pool	64	2700	-	-	2.703 milliseconds
asyncio_pool	64	10000	-	-	40.859 milliseconds
bare_thread	1	40	92.74 microseconds	64.52%	51.01 microseconds
bare_thread	1	300	105.44 microseconds	26.40%	293.91 microseconds
bare_thread	1	2700	112.38 microseconds	4.18%	2.578 milliseconds
bare_thread	1	10000	124.95 microseconds	1.31%	9.427 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
bare_thread	4	40	107.74 microseconds	67.82%	51.12 microseconds
bare_thread	4	300	120.65 microseconds	29.04%	294.76 microseconds
bare_thread	4	2700	123.95 microseconds	4.60%	2.569 milliseconds
bare_thread	4	10000	-	-	14.787 milliseconds
bare_thread	16	40	-	-	51.50 microseconds
bare_thread	16	300	-	-	295.16 microseconds
bare_thread	16	2700	-	-	2.578 milliseconds
bare_thread	16	10000	-	-	15.061 milliseconds
bare_thread	64	40	-	-	50.98 microseconds
bare_thread	64	300	-	-	296.18 microseconds
bare_thread	64	2700	-	-	2.627 milliseconds
bare_thread	64	10000	-	-	14.661 milliseconds
process_pool	1	40	282.18 microseconds	83.87%	54.25 microseconds
process_pool	1	300	299.03 microseconds	49.32%	307.25 microseconds
process_pool	1	2700	342.47 microseconds	11.36%	2.673 milliseconds
process_pool	1	10000	319.72 microseconds	3.30%	9.383 milliseconds
process_pool	4	40	99.69 microseconds	51.69%	93.17 microseconds
process_pool	4	300	-	-	569.80 microseconds
process_pool	4	2700	-	-	4.906 milliseconds
process_pool	4	10000	-	-	17.803 milliseconds
process_pool	16	40	-	-	91.27 microseconds
process_pool	16	300	-	-	583.13 microseconds
process_pool	16	2700	-	-	9.667 milliseconds
process_pool	16	10000	-	-	52.111 milliseconds
process_pool	64	40	-	-	89.02 microseconds
process_pool	64	300	-	-	574.61 microseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
process_pool	64	2700	-	-	27.072 milliseconds
process_pool	64	10000	-	-	160.095 milliseconds
thread_pool	1	40	30.68 microseconds	40.99%	44.16 microseconds
thread_pool	1	300	41.87 microseconds	12.38%	296.39 microseconds
thread_pool	1	2700	46.97 microseconds	1.76%	2.622 milliseconds
thread_pool	1	10000	61.46 microseconds	0.64%	9.515 milliseconds
thread_pool	4	40	46.09 microseconds	48.45%	49.03 microseconds
thread_pool	4	300	52.58 microseconds	14.55%	308.89 microseconds
thread_pool	4	2700	59.67 microseconds	2.23%	2.616 milliseconds
thread_pool	4	10000	-	-	19.820 milliseconds
thread_pool	16	40	-	-	49.32 microseconds
thread_pool	16	300	-	-	298.88 microseconds
thread_pool	16	2700	-	-	2.704 milliseconds
thread_pool	16	10000	-	-	26.429 milliseconds
thread_pool	64	40	-	-	46.87 microseconds
thread_pool	64	300	-	-	299.35 microseconds
thread_pool	64	2700	-	-	2.683 milliseconds
thread_pool	64	10000	-	-	56.167 milliseconds
zoo	1	40	534.31 microseconds	88.13%	71.94 microseconds
zoo	1	300	539.47 microseconds	62.15%	328.48 microseconds
zoo	1	2700	556.01 microseconds	17.04%	2.707 milliseconds
zoo	1	10000	554.64 microseconds	5.28%	9.951 milliseconds
zoo	4	40	547.26 microseconds	89.54%	63.91 microseconds
zoo	4	300	544.31 microseconds	62.78%	322.67 microseconds
zoo	4	2700	649.15 microseconds	19.63%	2.658 milliseconds

scheme	concurrency	body tier (microseconds)	framework overhead	overhead share	body as measured
zoo	4	10000	-	-	14.948 milliseconds
zoo	16	40	-	-	65.98 microseconds
zoo	16	300	-	-	317.65 microseconds
zoo	16	2700	-	-	2.642 milliseconds
zoo	16	10000	-	-	14.780 milliseconds
zoo	64	40	-	-	55.17 microseconds
zoo	64	300	-	-	321.35 microseconds
zoo	64	2700	-	-	2.753 milliseconds
zoo	64	10000	-	-	15.234 milliseconds

Appendix: Throughput and concurrency scaling

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
apscheduler	1	40	3404.3
apscheduler	1	300	1788.8
apscheduler	1	2700	334.5
apscheduler	1	10000	101.1
apscheduler	4	40	14970.0
apscheduler	4	300	7604.4
apscheduler	4	2700	1331.9
apscheduler	4	10000	394.8
apscheduler	16	40	70711.5
apscheduler	16	300	33030.1
apscheduler	16	2700	5375.2
apscheduler	16	10000	1548.6
apscheduler	64	40	285777.7
apscheduler	64	300	126173.2
apscheduler	64	2700	21702.3
apscheduler	64	10000	6046.2
asyncio_pool	1	40	6873.2
asyncio_pool	1	300	2564.2
asyncio_pool	1	2700	382.1
asyncio_pool	1	10000	103.0
asyncio_pool	4	40	26136.1
asyncio_pool	4	300	9344.3
asyncio_pool	4	2700	1428.5
asyncio_pool	4	10000	412.6
asyncio_pool	16	40	115629.2
asyncio_pool	16	300	40462.9

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
asyncio_pool	16	2700	5831.5
asyncio_pool	16	10000	1605.9
asyncio_pool	64	40	530094.2
asyncio_pool	64	300	172085.1
asyncio_pool	64	2700	22599.7
asyncio_pool	64	10000	6281.4
bare_thread	1	40	6956.3
bare_thread	1	300	2504.1
bare_thread	1	2700	371.7
bare_thread	1	10000	104.7
bare_thread	4	40	25180.5
bare_thread	4	300	9629.0
bare_thread	4	2700	1485.5
bare_thread	4	10000	405.2
bare_thread	16	40	93082.7
bare_thread	16	300	37756.1
bare_thread	16	2700	5917.6
bare_thread	16	10000	1604.5
bare_thread	64	40	376362.9
bare_thread	64	300	150577.9
bare_thread	64	2700	23154.0
bare_thread	64	10000	6718.2
process_pool	1	40	2972.4
process_pool	1	300	1649.4
process_pool	1	2700	331.7
process_pool	1	10000	103.1
process_pool	4	40	20740.7
process_pool	4	300	14093.4
process_pool	4	2700	2878.6
process_pool	4	10000	868.1
process_pool	16	40	99052.0
process_pool	16	300	57486.6
process_pool	16	2700	11345.1
process_pool	16	10000	3451.0
process_pool	64	40	325635.1
process_pool	64	300	212645.4
process_pool	64	2700	47269.4
process_pool	64	10000	13826.1
thread_pool	1	40	13362.6
thread_pool	1	300	2956.3
thread_pool	1	2700	374.6
thread_pool	1	10000	104.4
thread_pool	4	40	42055.6
thread_pool	4	300	11065.7

scheme	concurrency	body tier (microseconds)	throughput (tasks/s)
thread_pool	4	2700	1494.9
thread_pool	4	10000	405.4
thread_pool	16	40	173329.2
thread_pool	16	300	45260.1
thread_pool	16	2700	5735.0
thread_pool	16	10000	1577.0
thread_pool	64	40	715882.7
thread_pool	64	300	180117.2
thread_pool	64	2700	22956.6
thread_pool	64	10000	6220.2
zoo	1	40	1649.5
zoo	1	300	1152.1
zoo	1	2700	306.5
zoo	1	10000	95.2
zoo	4	40	6544.9
zoo	4	300	4613.7
zoo	4	2700	1209.4
zoo	4	10000	373.8
zoo	16	40	26218.8
zoo	16	300	18208.4
zoo	16	2700	4839.3
zoo	16	10000	1510.0
zoo	64	40	113090.9
zoo	64	300	75956.6
zoo	64	2700	19258.5
zoo	64	10000	6017.5