Performance Analysis Report

The test script ran 12 tests across 2 test suites, with all tests passing. The tests included both API functionality and simulation.

Table 1: Resource Usage Analysis

This table shows how four different resources behaved across four test runs.

IdentityProviderClient stayed perfectly flat at 3,840,256 units across all runs. No variation at all. This resource gets used the exact same way every time. The percentage hovered around 69%, meaning it accounts for most of the resource usage we're measuring.

SmartMachineClient also never changed. Stayed at 32,256 units for all four runs. The percentage was 0.58% each time. This is a much smaller piece of overall resource use compared to IdentityProviderClient.

MachineResourceInfo locked in at 1,675,064 units. Again, zero variation across runs. It represented about 30% of total resource usage. Second biggest resource after IdentityProviderClient.

DataCache held steady at 23,696 units. No changes between runs. The percentage was 0.43% every time. Smallest resource usage of the four.

Every single metric came back identical across all four runs. This is unusual compared to the earlier tables where we saw small variations. It suggests these resources follow a completely deterministic path. The algorithm makes the exact same calls to these clients and caches regardless of what else varies in the system.

The percentages tell us IdentityProviderClient and MachineResourceInfo dominate resource usage. Together they account for roughly 99% of what's measured here. SmartMachineClient and DataCache barely register.

(index)	Resource	Run 1 Units	Run 1 %	Run 2 Units	Run 2 %	Run 3 Units	Run 3 %	Run 4 Units	Run 4 %
0	'IdentityProviderClient'	3840256	'69.39%'	3840256	'69.26%'	3840256	'69.33%'	3840256	'68.93%'
1	'SmartMachineClient'	32256	'0.58%'	32256	'0.58%'	32256	'0.58%'	32256	'0.58%'
2	'MachineStateTable'	1675064	'30.27%'	1675064	'30.21%'	1675064	'30.24%'	1675064	'30.07%'
3	'DataCache'	23696	'0.43%'	23696	'0.43%'	23696	'0.43%'	23696	'0.43%'

Table 2: Cache Performance

Cache hits went from 3,812 in Run 1 down to 3,678 in Run 4. That's 134 hits of difference, about 3.6% variation. Run 1 performed best. Run 4 performed worst.

Cache misses ranged from 2,100 to 2,224. The pattern flipped from cache hits. When hits were high, misses were low. When hits dropped, misses went up.

Hit rate stayed between 62% and 64%. Run 1 got 64.48%. Run 4 got 62.32%.

Total memory accesses (hits plus misses) barely changed. We saw between 5,902 and 5,912 total accesses. The algorithm follows mostly the same path each time but with small variations in how it moves through memory.

A 63% hit rate means one in three memory accesses misses the cache. That's okay but not great.

(index)	Run	Cache Hits	Cache Misses	Hit Rate
0	1	3812	2100	'64.48%'
1	2	3779	2201	'63.19%'
2	3	3797	2129	'64.07%'
3	4	3678	2224	'62.32%'

Table 3: Database Access

Cache hits match Table 2 exactly. Same test runs, same data.

DB accesses never changed. All four runs hit exactly 6,822 database calls. Most stable number across all three tables. The algorithm makes the same database calls every single time.

Hit/Access ratio went from 0.5588 down to 0.5391. This divides cache hits by database accesses. For every database call, we get about 0.55 cache hits on average.

The ratio sitting at 0.55 (compared to the 63% cache hit rate) tells us database calls aren't the only memory operations happening. Other stuff is accessing memory too.

Database accesses stay locked at 6,822 while cache hits vary by 134. The variation in cache performance comes from operations outside the database layer or from how we handle database results in memory.

(index)	Run	Cache Hits	DB Accesses	Hit/Access Ratio
0	1	3812	6822	'0.5588'
1	2	3779	6822	'0.5539'
2	3	3797	6822	'0.5566'
3	4	3678	6822	'0.5391'