WHITE PAPER

The ROI of Shared Agentic Memory

How Spark Reduces Cost, Saves Time, and Makes AI Coding Agents Predictable

MemCo – The Memory Company
March 2026

spark.memco.ai

Executive Summary

40%	34%	31%
Cost Reduction	Time Savings	Fewer Steps

For a 10-developer team using AI coding agents, Spark's shared agentic memory reduces LLM costs by 40%, frees approximately 230 hours of developer capacity per month, and makes agent behaviour predictable and budgetable — with measurable returns beginning from the second pass through similar problems.

These results come from a controlled experiment in which we ran the same 10 SWE-bench bug-fixing tasks six times each under two conditions: without Spark (baseline) and with Spark (cumulative shared memory). Table 1 summarises the key results.

Table 1. Summary of key results.

Metric	Without Spark	With Spark (Steady State)	Improvement
Avg Cost (10 tasks)	$21.85	$13.14	40% lower
Avg Duration	56.4 min	37.1 min	34% faster
Avg Steps	1,727	1,190	31% fewer
Pass Rate	65%	75%	+10pp

The gains extend across the full difficulty spectrum. On tasks the agent already solved 100% of the time, Spark still reduced cost by 35%. On a task the agent never solved in any condition, Spark reduced the cost of failure by 34%. Regression analysis confirms that the improvements are driven by the specific quality of knowledge retrieved from memory: each piece of knowledge the agent finds relevant is associated with $0.34 in cost savings and 23 fewer agent steps ($p < 0.001$).

Critically, Spark delivers value fast. The first run seeds the shared memory at no additional cost. By the third run, costs have already fallen 40% below baseline and remain there. The adoption curve is measured in days, not months.

The Problem

AI coding agents in production today are fundamentally amnesiac. Every task starts from a blank slate. Solutions discovered in one session are forgotten before the next one begins. This creates three compounding inefficiencies.

Wasted compute. Agents re-derive solutions that have already been found, incurring the same token cost repeatedly for the same category of problem.

Unpredictable behaviour. Because the path to a solution is always reconstructed from scratch, some runs find the optimal path quickly while others take expensive detours through dead ends. The result is high variance in both cost and time, making sprint planning unreliable.

No collective learning. Insights discovered by one agent are never shared with others, preventing the emergence of the kind of collective intelligence that makes human engineering teams more than the sum of their parts.

Spark addresses this gap. It provides a persistent, curated, shared memory layer that agents interact with via standard MCP (Model Context Protocol) tool calls. As agents work, they retrieve relevant knowledge from the shared memory and contribute new insights back. The memory grows with use, and the benefits compound.

The Experiment

We selected 10 bug-fixing tasks from the SWE-bench benchmark, spanning five Python repositories (pydata/xarray, pytest-dev/pytest, scikit-learn/scikit-learn, sphinx-doc/sphinx, and sympy/sympy). Each task requires the agent to diagnose a real bug and produce a correct fix, verified by the repository's own test suite. Three tasks are solved in all conditions, and one is never solved. We consider this representative of real-world workloads, which typically contain a mix of routine, challenging, and beyond-capability tasks.

Table 2. Experimental parameters.

Parameter	Value
Task Source	SWE-bench verified (10 tasks from 5 Python repositories)
Model / Harness	Claude Sonnet 4.5 / Claude Code
Runs	6 without Spark (baseline) + 6 with Spark (cumulative memory)
Metrics	Pass/fail, wall-clock duration, agent steps, token usage, dollar cost

The baseline runs are independent: each starts from scratch with no memory. The Spark runs are sequential: each run benefits from knowledge accumulated by all previous runs, modelling a team whose collective knowledge grows with every task completed.

The first Spark run is the cold start. The shared memory is empty; the agent works through each task and populates the memory with initial insights. From the second run onward, the agent retrieves knowledge from the memory as it works, and contributes further insights when it discovers something new. This models the real-world adoption path: install Spark, let agents work normally, and the shared memory begins building itself from the very first task.

The Business Case for Spark

Direct Cost Savings

The most immediate ROI comes from reduced token consumption. At steady state, the average cost per 10-task run drops from $21.85 to $13.14, a 40% reduction. Table 3 projects the savings for a 10-developer team.

Table 3. Projected monthly cost[1] savings for a 10-developer team.

Scenario	Without Spark	With Spark
Cost per 10-task run	$21.85	$13.14
4 runs per developer per day	$87.40	$52.56
Monthly cost per developer	$1,748	$1,051
Team of 10, monthly	$17,480	$10,512
Monthly savings	—	**$6,968 / month**

At steady-state Spark adoption, the projected saving is approximately $7,000 per month for a 10-developer team, or over $80,000 annually. These are direct, measurable savings on the LLM bill.

Regression analysis (as detailed in the Appendix) quantifies the mechanism: each piece of knowledge the agent received from Spark and finds relevant is independently associated with $0.34 in cost savings and 23 fewer agent steps ($p < 0.001$). At steady state, agents receive an average of 2–3 relevant pieces of knowledge per task. The savings are not an artefact of having Spark enabled; they are driven by the specific quality of the knowledge retrieved.

Developer Time and Opportunity Cost

The 34% reduction in wall-clock time is arguably more valuable than the direct cost savings, particularly in organisations where developer time is the binding constraint.

Table 4. Projected developer time savings for a 10-developer team.

Scenario	Without Spark	With Spark
Duration per 10-task run	56.4 min	37.1 min
4 runs per developer per day	3h 46min	2h 29min
Time saved per dev per day	—	1h 17min
Team of 10, monthly savings	—	**~257 hours**

Spark frees approximately 257 hours of developer capacity per month across a 10-person team. This is time that can be redirected to code review, architecture, design, or the kind of creative work that agents are not yet capable of.

---

[1] Costs are estimated using the published Anthropic API usage costs. Usage costs based on subscriptions will vary. Reference: https://platform.claude.com/docs/en/build-with-claude/prompt-caching#pricing, March 2nd 2026.

Parnin and Rugaber[2] studied 10,000 programming sessions and found that after an interruption, 90% of developers could not resume coding within one minute; most needed to navigate to multiple code locations to rebuild their mental model of the problem. Every question that an agent can answer from shared memory, instead of requiring a developer to context-switch away from their current task, preserves that mental model.

Predictable Budgets, Predictable Sprints

Without Spark, cost per run ranged from $20.11 to $23.24 (standard deviation $1.36). At steady state, cost ranged from $12.37 to $13.63 (standard deviation $0.55). Not only is the mean substantially lower, but the variance has more than halved. A team that can reliably forecast how long an agent-assisted sprint will take, and how much it will cost, can allocate capacity, set budgets, and make delivery commitments with confidence.

The Evidence

From Seed to Steady State

The resource savings track the maturity of the shared memory. We observe three distinct phases in the data.

Table 5. Resource efficiency by phase.

Phase	Runs	Avg Steps	Avg Duration	Avg Cost	Cost vs Baseline
No Spark (baseline)	6	1,727	56.4 min	$21.85	—
Seed (run 1)	1	1,775	52.8 min	$22.46	+3%
Early returns (run 2)	1	1,504	49.1 min	$20.38	−7%
Steady state (runs 3–6)	4	1,190	37.1 min	**$13.14**	**−40%**

The seed phase is run 1 — the first time the agents encounter these tasks with Spark enabled. The shared memory is empty. The agent works normally, and its experience populates the memory: 9 new insights are generated across the 10 tasks. Cost and steps are indistinguishable from baseline. From the agent's perspective, nothing has changed — but the memory is now seeded.

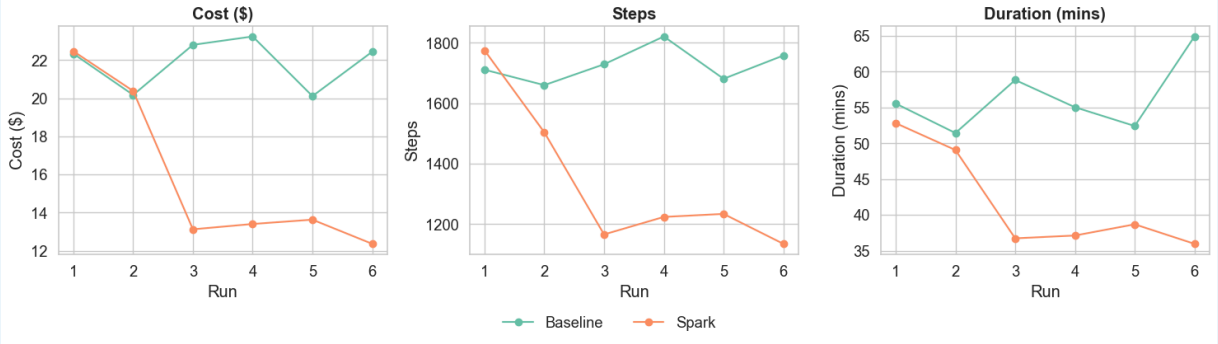
The early returns phase is run 2. The agents now receive knowledge from the memory as they work. Every task receives at least one piece of knowledge that the agent finds relevant. The cost begins to drop, and the agent takes 15% fewer steps. The memory is still young; 3 more new insights are contributed as the agents encounter situations not yet covered.

The steady state arrives at run 3. Cost drops 40% below baseline and stays there through run 6. The memory has sufficient coverage that new insights become rare — only 4 are generated across runs 3 to 6 — while the volume of relevant knowledge delivered to agents grows steadily from 16 retrievals in run 3 to 26 in run 6. There is less new to discover, but more useful knowledge being surfaced.

---

[2] Parnin, C. & Rugaber, S. (2011). "Resumption Strategies for Interrupted Programming Tasks." Software Quality Journal, 19(1), 5–34. https://doi.org/10.1007/s11219-010-9104-9

Figure 1: Resource use across 6 baseline and 6 Spark runs.

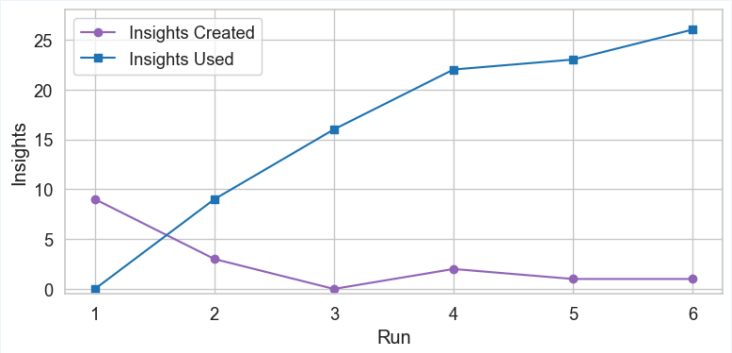


Baseline: higher resource use throughout, not improving over time.

Spark: costs start similar to baseline at seed stage (run 1), dropping sharply at steady state (from run 3). Steps and duration showing a similar pattern.

The practical implication is a short adoption curve. Spark does not require weeks of tuning or a dedicated onboarding project. The agents seed the memory by doing their normal work. By the third pass through similar problem types, the shared memory is delivering its full efficiency benefit.

Figure 2: Knowledge adoption curve



*Spark **early stages** (runs 1-2) are about efficient knowledge collection.*

*At **steady-state** (runs 3 - onwards) the system shifts into leveraging the knowledge already collected into the shared memory, and less new knowledge is discovered.*

Benefits Across the Difficulty Spectrum

A natural question is whether shared memory only matters for certain kinds of problems. The data shows that Spark delivers value across the full difficulty spectrum — from tasks the agent always solves, through borderline tasks, to tasks beyond the model's current capability.

Always-solved tasks: faster and cheaper

The three pytest tasks were solved in 100% of runs in both conditions. The pass rate was always perfect. Yet the efficiency differences are striking.

Table 6. Efficiency on always-solved tasks (pytest, 100% pass rate in all conditions).

Metric (pytest tasks only)	No Spark	Steady-State Spark	Improvement
Pass Rate	100%	100%	(unchanged)
Avg Number of Steps (3 tasks)	298	227	24% fewer
Avg Cost (3 tasks)	$3.54	$2.32	34% lower
Avg Duration (3 tasks)	8.4 min	6.8 min	19% faster

On tasks the agent was already solving reliably, Spark reduced costs by 34% and steps by 24%. The agent did not solve the problems more often; it solved them more efficiently. This matters because the bulk of any development team's workload consists of routine tasks. If shared memory can cut the cost of solved problems by over a third, the savings compound rapidly across thousands of agent-assisted tasks per month.

Never-solved tasks: failing faster and cheaper

At the other end of the spectrum, the scikit-learn task was never solved in any condition — baseline or Spark. This problem sits beyond the capability ceiling of Sonnet 4.5 and, without human help, the problem remains unsolved. Regardless, the resource consumption still improves with Spark.

Table 7. Efficiency on the never-solved task (203-sklearn, 0% pass rate in all conditions).

Metric	No Spark	With Spark (all runs)	Improvement
Pass Rate	0%	0%	(unchanged)
Avg Steps	202	154	15% fewer
Avg Cost	$3.24	$2.14	34% lower

Even in failure, the agent used 15% fewer steps and cost 34% less when Spark was available. The agent still failed, but it failed faster and cheaper. It explored fewer dead ends before recognising that its approaches were not converging.

This matters more than it might seem. In production, not every task an agent attempts will be solvable. When the agent fails, a developer must step in. The cost of that failure is the sum of the wasted compute and the developer time spent diagnosing what went wrong. By shortening the failed attempt, shared memory reduces both components. The agent produces a shorter, more focused trace for the developer to review, and the compute budget consumed before escalation is lower.

The full picture

The combination of these cases completes the picture. Costs decrease when Spark is available regardless of the outcome category: for simple problems that are always solved, for borderline problems whose reliability improves with accumulated knowledge, and even for hard problems that the agents never quite manage to crack. The mechanism is the same in each case: the shared memory

gives the agent a shorter path through the problem space, whether that path leads to a solution or to a faster recognition that the problem is beyond reach.

The accumulation of knowledge in the shared memory also tends to make agent behaviour more stable. Agents take fewer steps to reach a solution or give up, and the outcome of borderline problems becomes less random when Spark is available. In four out of six variable-outcome tasks, we observe a clear transition point after which the task is solved in every subsequent run — a pattern absent from the baseline, where outcomes remain noisy throughout.

Discussion

Knowledge Efficiency and Compound Returns

One of Spark's most valuable properties is that its ROI compounds over time. Across the full experiment, 16 new insights were generated — 9 during the seed run and only 7 across the subsequent five runs. Yet the volume of relevant knowledge delivered to agents grew from 0 in the seed run to 26 by run 6. A small number of insights, discovered early, proved useful across a wide range of subsequent tasks.

Spark is designed to de-duplicate knowledge. When an agent contributes an insight that is semantically equivalent to one already in the memory, Spark recognises the duplication and does not store it as new. The insight counts in our data reflect only genuinely novel contributions to the shared memory. The declining rate of new insights is not a sign of diminishing returns — it is a sign that the memory's coverage of the problem space is converging. And as coverage converges, the frequency of relevant knowledge retrievals rises.

This compounding property distinguishes Spark from one-time optimisation efforts such as prompt engineering or fine-tuning. A prompt that works well today may need revision as models update. A fine-tuned model may lose its advantage when the base model is replaced. Spark's memory is model-agnostic: the curated insights describe problems and solutions in natural language, not model-specific patterns, and they retain their value across model upgrades.

Conclusion

The data from this experiment supports four conclusions about the business value of shared agentic memory.

Shared memory reduces cost and time — even on problems already solved. Steady-state Spark adoption reduced costs by 40%, time by 34%, and steps by 31%. On tasks the agent already solved 100% of the time, cost still fell by 34%. On a task the agent never solved, the cost of failure fell by 34%. The savings come not from solving more problems, but from solving them — or failing at them — with a shorter path.

Knowledge capture is fast; returns follow quickly. A single seed run populates the memory at no additional cost. By the third run, the full efficiency benefit has arrived. The adoption curve is measured in days, not months. Agents need no special configuration — they seed the memory simply by doing their normal work.

The system learns efficiently and compounds. Only 16 insights were needed to drive the full improvement. Those insights were retrieved 96 times across 60 task-level attempts. The ROI improves with every run.

Agent behaviour becomes predictable and budgetable. Cost variance more than halved. Agents take fewer steps to reach a solution or recognise that a problem is beyond reach. Sprint outcomes become forecastable. This is a prerequisite for scaling agent use across an engineering organisation.

Together, these findings point to a simple economic argument. For a 10-developer team, Spark's direct cost savings exceed $83,000 per year. The time savings free over 250 hours of developer capacity per month. The improved stability and reduced variance lower the human rework burden and make sprint outcomes predictable. And because the shared memory compounds in value over time, the return on investment improves with every run.

The next frontier of AI-assisted development is not just about bigger models. It is about agents that learn from experience, share what they learn, and get better at helping your team every day.

Appendix: Isolating the Mechanism

The main body of this paper demonstrates that agents with Spark use fewer resources and solve more problems. This appendix presents regression analysis isolating the effect of receiving relevant knowledge, as opposed to merely having Spark enabled.

At the end of each task, the agent sends feedback to Spark indicating whether the knowledge it received was relevant or not. We ran OLS regressions with cluster-robust standard errors (clustered on problem) across 120 observations. The independent variable was the count of insights the agent found relevant; the dependent variables were steps, duration, and cost.

Table A1. OLS regression: impact of relevant knowledge on resource use.

Metric	β per relevant insight	95% CI	p-value
Steps	−23.0	[−34.8, −11.3]	0.0001
Duration (hours)	−0.014	[−0.022, −0.005]	0.002
Cost ($)	−0.339	[−0.477, −0.201]	< 0.001

All three resource metrics show a statistically significant negative association with receiving relevant knowledge ($p < 0.01$). Each piece of relevant knowledge is associated with approximately 23 fewer steps, 0.8 fewer minutes, and $0.34 in cost savings. The confidence intervals are comfortably away from zero, confirming that the improvements are driven by the specific content of the retrieved knowledge.

We also tested whether receiving relevant knowledge improves the probability of solving a problem.

Table A2. Solve rate by knowledge relevance.

Condition	Solve Rate	n
No relevant knowledge received	65.3%	72
Relevant knowledge received	70.8%	48
Odds ratio per relevant insight	**1.35×**	**p = 0.049**

The observed solve rate is 70.8% when the agent receives relevant knowledge, compared to 65.3% when it does not. The odds ratio of 1.35× reaches statistical significance at the 5% level ($p = 0.049$, 95% CI: 1.00–1.82).

When restricted to just the six variable-outcome tasks (those that were sometimes solved, sometimes not) the effect is greater and more significant (OR 1.76×, $p < 0.001$).