# Spark Mode Efficiency: Direct Edit vs Proposal Mode

Benchmarking Spark subagent write modes under GPT-5.5 coordinators

By Adam Owada, with Codex

June 25, 2026

---

## Abstract

This paper evaluates whether `gpt-5.3-codex-spark` coding subagents are more effective when they directly edit code or when they only propose changes for a `gpt-5.5` coordinator to implement. The benchmark is RuleLedger v3 (RuleLedger v3 white paper), a TypeScript and Python software-engineering task built around event replay, billing semantics, cross-language parity, hidden tests, performance checks, and maintainability review.

For RuleLedger v3, adding six xhigh Spark leaves did not improve the best quality or efficiency result. Solo GPT-5.5 xhigh produced the highest observed mean quality (`0.755`). Solo GPT-5.5 high beat medium direct Spark on quality (`0.696` vs `0.622`) and aggregate GPT-token efficiency (`2.66e-7` vs `2.55e-7` quality per GPT token). Medium direct Spark beat medium solo (`0.622` vs `0.462`), but required 75% more GPT tokens and 176% more total implementation tokens than medium solo. When high reasoning is available, medium direct Spark is an expensive partial catch-up.

Among Spark-assisted runs, direct edit had higher observed mean composite quality than proposal mode at all four GPT-5.5 coordinator reasoning levels, with the largest gap at medium (`+0.093` quality). Direct edit also used fewer total implementation tokens than proposal mode at low, medium, and high. Proposal mode fits read-only review or governance workflows, but did not produce a token-efficiency win here.

## Decision Summary

| Developer goal | Recommended configuration | Why |
|---|---|---|
| Maximize code quality | Solo GPT-5.5 xhigh | Highest observed mean quality: `0.755` |
| Balance strong quality with aggregate GPT-token efficiency | Solo GPT-5.5 high | Higher quality and better aggregate GPT-token efficiency than medium direct Spark |
| Improve a medium-only coordinator while using separate Spark capacity | Medium root + direct Spark leaves | Best same-level Spark lift: `+0.160` quality over medium solo |
| Use Spark leaves for implementation | Direct-edit Spark leaves | Better observed quality than proposal at every coordinator reasoning level |
| Keep Spark leaves read-only | Proposal-only Spark leaves | Fits review, governance, or separate-approval workflows |

Medium direct Spark is the only Spark-assisted configuration that materially changes the same-level result: `+0.160` over medium solo. High solo remains stronger when high reasoning is available.

## Terminology

| Term | Meaning |
|---|---|
| GPT-5.5 coordinator | The root agent. It plans the run, integrates subagent outputs, and owns the final submitted implementation. |
| Spark leaf | A `gpt-5.3-codex-spark` worker assigned one implementation or review slice. Spark leaves never call other agents in this experiment. |
| Reasoning level | A runtime reasoning-effort setting in the benchmark harness: `low`, `medium`, `high`, or `xhigh`. |
| Direct edit mode | Spark leaves work in separate writable git worktrees and return diffs for the coordinator to inspect. The leaf prompts call these isolated leaf workspaces; git worktree is the underlying mechanism. |
| Proposal mode | Spark leaves run read-only and return findings, proposed patches, tests, and integration notes. |
| GPT tokens | GPT-5.5 implementation tokens. For Spark-assisted runs, this is coordinator planning plus coordinator integration. |
| Spark tokens | Spark leaf implementation tokens. |
| Total implementation tokens | GPT implementation tokens plus Spark implementation tokens. Judge tokens are excluded from efficiency tables. |

GPT-5.5 and Spark are different models with different roles in this experiment. GPT-5.5 is the root/coordinator model and also the solo model. Spark is the leaf model. Spark usage is tracked separately because it may draw from a different budget, quota, or price schedule than GPT-5.5. The paper reports GPT tokens, Spark tokens, and total implementation tokens so readers can apply their own cost model.

## Benchmark Task

RuleLedger v3 is a subscription-ledger implementation benchmark. The model must read event streams, normalize events, replay account state, compute billing and entitlements, export deterministic reports, and preserve compatibility with earlier public APIs. The task has both TypeScript and Python surfaces, and the two implementations must agree on behavior.

The difficult part is global consistency. A local fix that passes one public test can still fail hidden cases if summaries, reports, billing, parity outputs, and replay digests use different state-transition logic. RuleLedger v3 therefore rewards implementations that build one coherent replay model and apply it across the whole codebase.

## Tested Topology

The Spark-assisted topology was staged:

```
Frozen starter project
  -> GPT-5.5 coordinator planning stage
  -> six Spark leaf stages
       - direct mode: separate writable git worktrees with diffs
       - proposal mode: read-only inspection with proposals
  -> GPT-5.5 coordinator integration stage
  -> public tests, hidden tests, scoring, and judge assessment
```

Each Spark-assisted run used exactly six xhigh Spark leaves:

| Leaf | Responsibility |
|---|---|
| 1 | TypeScript parsing, normalization, views, and migration compatibility |
| 2 | TypeScript replay, billing, reporting, performance, and API integration |
| 3 | Python parsing, normalization, views, and migration compatibility |
| 4 | Python replay, billing, reporting, performance, and API integration |
| 5 | Cross-language parity, fixtures, public tests, and regression review |

| Leaf | Responsibility |
|---|---|
| 6 | Adversarial review for localization, maintainability, performance, and hidden-test risk |

The direct-edit prompt gave leaves write permission:

```
You are in direct edit mode inside an isolated leaf workspace. You may edit
files in this copy. The root integration run will inspect your diff and decide
what to land in the final measured workspace.
```

The proposal-only prompt kept leaves read-only:

```
You are in proposal mode. Do not edit files. Inspect the visible workspace and
return concrete findings, proposed patches, tests, and integration notes.
```

The implementation submitted for scoring was always the final workspace produced by the GPT-5.5 coordinator integration stage.

## Experiment Design

The official comparison set contains `360` scored implementation runs:

| Group | Cells | Runs per cell | Rows |
|---|---|---|---|
| Solo GPT-5.5 | 4 reasoning levels | 50 | 200 |
| Spark direct | 4 reasoning levels | 20 | 80 |
| Spark proposal | 4 reasoning levels | 20 | 80 |

The final analysis artifact ingested `384` rows for audit. The extra `24` rows are pilot and bridge diagnostics and are excluded from the headline comparison tables.

Spark-assisted variables:

- GPT-5.5 coordinator reasoning: `low`, `medium`, `high`, `xhigh`.
- Spark leaf model: `gpt-5.3-codex-spark`.
- Spark leaf reasoning: `xhigh`.
- Spark leaf count: six.
- Spark mode: direct edit or proposal-only.
- Judge: GPT-5.5 xhigh.

Solo comparison group:

- GPT-5.5 solo runs from the RuleLedger v3 benchmark testing.
- Reasoning levels: `low`, `medium`, `high`, `xhigh`.
- Runs per reasoning level: 50.

## Measurement

Reported values use fixed rounding: quality-like scores, deltas, intervals, and component means are rounded to three decimals; token counts are shown as whole-token means or `M` shorthand in prose; efficiency ratios use three significant figures in scientific notation.

### Quality

Quality is a 0 to 1 composite score from `configs/scoring_v3.yaml`:

| Component | Weight | Meaning |
|---|---|---|
| Hidden correctness | 0.50 | Correctness on frozen private RuleLedger v3 cases, excluding parity and performance categories |
| Hidden parity | 0.15 | TypeScript/Python agreement on hidden parity checks |
| Performance | 0.10 | Performance behavior on hidden workloads |
| Judge | 0.22 | GPT-5.5 xhigh assessment of source, diffs, logs, public checks, and hidden-result summaries |
| Minimality | 0.03 | Production LOC minimality signal relative to the configured target |

The judge was topology-blind: it was instructed not to infer or reward the producing agent arrangement. It saw implementation evidence, public logs, diffs, and hidden-result summaries. It did not see private hidden case payloads. If the judge returned numeric `correctness_score`, `parity_score`, `maintainability_score`, and `test_evidence_score` fields, the harness averaged those four fields. If strict JSON parsing failed, the judge component scored zero.

## Tokens

The harness parsed implementation usage from `codex exec --json` event streams. Official Spark-assisted rows use `usage_attribution_method=per_event_model`: JSONL events carried model metadata, and the analysis attributed GPT-5.5 coordinator tokens and Spark leaf tokens from those events. Solo comparison rows use total implementation usage as GPT-5.5 usage. Judge tokens are excluded from the efficiency tables because the measured developer decision is which implementation topology to run, not how expensive the benchmark judge was.

The paper reports:

- `GPT tokens`: GPT-5.5 coordinator implementation tokens.
- `Spark tokens`: Spark leaf implementation tokens.
- `Total tokens`: GPT implementation tokens plus Spark implementation tokens.
- `Quality/GPT token`: mean quality divided by mean GPT implementation tokens.
- `Quality/total token`: mean quality divided by mean total implementation tokens.

## Efficiency Estimands

Two efficiency estimands are useful, and they answer different questions:

| Metric | Formula | Question it answers |
|---|---|---|
| Ratio of means | `mean(quality) / mean(tokens)` | If this configuration is run many times, how much quality did the aggregate token budget buy? |
| Mean of ratios | `mean(quality / tokens)` | How efficient was the average individual run? |

This paper uses ratio-of-means as the primary efficiency metric because it matches aggregate budget planning. Mean-of-ratios is reported as a sensitivity check where it changes interpretation. The generated analysis artifacts include fields named `quality_per_*_mean`; those are means of per-run ratios. The tables in this paper recompute ratio-of-means from the displayed means so the math is directly reproducible.

The distinction matters for high solo versus medium direct:

| Comparison | High solo | Medium direct | Winner |
|---|---|---|---|
| Ratio of means | `2.66e-7` | `2.55e-7` | High solo |
| Mean of ratios | `2.88e-7` | `3.00e-7` | Medium direct |

When the winner changes with the estimand, the comparison is metric-sensitive. For aggregate GPT-token planning, high solo has higher quality and a narrow aggregate-efficiency lead.

### Uncertainty

The paper reports observed sample means, medians, standard deviations, and approximate bootstrap intervals for key direct-versus-proposal deltas. Spark-assisted cells have 20 runs each, so small observed gaps should be read cautiously. The most robust Spark write-mode result is medium direct over medium proposal; low and xhigh direct-versus-proposal gaps are small.

## Results at a Glance

| Root reasoning | Solo quality / total tokens | Direct quality / total tokens | Proposal quality / total tokens | Primary read |
|---|---|---|---|---|
| low | `0.434` / `0.815M` | `0.482` / `2.917M` | `0.472` / `3.100M` | Spark raises observed mean quality slightly, at high token cost |
| medium | `0.462` / `1.390M` | `0.622` / `3.843M` | `0.529` / `4.320M` | Largest Spark lift, still below high solo |
| high | `0.696` / `2.612M` | `0.712` / `3.433M` | `0.652` / `4.497M` | Direct leads Spark modes; solo high uses fewer tokens |
| xhigh | `0.755` / `3.334M` | `0.734` / `4.488M` | `0.714` / `4.430M` | Solo xhigh has the highest mean |

## Detailed Findings

### Low Reasoning

Low direct had higher observed mean composite quality than low solo: `0.482` versus `0.434`. Low proposal scored `0.472`. The direct-vs-proposal quality gap was small (`+0.010` for direct), and low proposal had the higher median quality and hidden correctness. The low result is weak evidence for direct edit.

Low Spark has a poor cost/quality profile. Low direct used `2.917M` total implementation tokens versus `0.815M` for low solo. The quality lift is real in the mean, but the configuration spends far more total tokens for a result that remains below medium direct, high solo, and xhigh solo.

### Medium Reasoning

Medium direct is the largest positive Spark case. It scored `0.622`, compared with `0.462` for medium solo and `0.529` for medium proposal. The direct-vs-proposal quality gap was `+0.093`, the largest direct-edit advantage in the experiment. Medium direct also used fewer GPT, Spark, and total implementation tokens than medium proposal.

The relevant comparison is medium direct versus high solo. Medium direct used `2.436M` GPT tokens, compared with `1.390M` for medium solo and `2.612M` for high solo. It used `3.843M` total implementation tokens, compared with `1.390M` for medium solo and `2.612M` for high solo. High solo scored higher quality (`0.696` vs `0.622`) and has the better ratio-of-means GPT efficiency (`2.66e-7` vs `2.55e-7`).

Medium direct is a same-level lift for medium-constrained workflows. High solo remains stronger when high reasoning is available.

### High Reasoning

High direct scored `0.712`, high proposal scored `0.652`, and high solo scored `0.696`. The observed mean favors high direct over high solo by `+0.016`, but that gap is small relative to run-to-run variance. High solo also uses fewer total tokens: `2.612M` versus `3.433M` for high direct.

The larger high-level result is within Spark modes: direct was better than proposal. High direct used about `132k` fewer GPT tokens, `932k` fewer Spark tokens, and `1.064M` fewer total tokens per run than high proposal while also scoring higher observed mean quality.

High solo has higher quality than medium direct and fewer total tokens than high direct. High direct may be worth testing in a workflow that already benefits from parallel Spark work, but this result does not make it the first choice.

### Xhigh Reasoning

Solo xhigh had the best observed mean quality in the study: `0.755`. Xhigh direct scored `0.734`, and xhigh proposal scored `0.714`. Neither Spark-assisted xhigh mode improved on solo xhigh.

Xhigh proposal saved about `242k` GPT tokens and `59k` total tokens relative to xhigh direct, but it did not save GPT tokens relative to xhigh solo. Proposal mode is therefore only attractive within an already Spark-assisted xhigh design where read-only leaves are valuable enough to justify lower observed quality.

The xhigh result shows the risk of decomposition on this task. RuleLedger v3 rewards a single coherent replay model across TypeScript, Python, billing, reports, summaries, and parity checks. At xhigh, the solo model appears to preserve that global coherence better than the tested Spark topology.

## Direct Edit vs Proposal Mode

Direct edit had higher observed mean composite quality than proposal mode at all four coordinator reasoning levels:

| Root | Direct minus proposal quality | Approx. bootstrap interval | Interpretation |
|---|---|---|---|
| low | `+0.010` | `[-0.080, +0.098]` | Small and noisy |
| medium | `+0.093` | `[+0.008, +0.177]` | Largest direct-edit advantage |
| high | `+0.060` | `[-0.035, +0.156]` | Directionally favors direct |
| xhigh | `+0.020` | `[-0.076, +0.113]` | Small and noisy |

Direct edit also used fewer tokens in most Spark-assisted cells. At low, medium, and high, direct edit used fewer GPT tokens, fewer Spark tokens, and fewer total implementation tokens than proposal mode. Proposal mode used more Spark tokens at every coordinator reasoning level.

| Root | GPT token delta, direct minus proposal | Spark token delta | Total token delta |
|---|---|---|---|
| low | `-41,883` | `-140,896` | `-182,779` |
| medium | `-72,324` | `-404,427` | `-476,751` |
| high | `-131,950` | `-931,645` | `-1,063,595` |
| xhigh | `+241,841` | `-183,146` | `+58,696` |

Choose direct edit when Spark leaves are expected to contribute implementation work and the coordinator can review diffs. Choose proposal mode when the leaves must remain read-only or when the purpose is independent review rather than token-efficient implementation.

## Why Direct Edit Helped

Direct edit gave the coordinator concrete code artifacts: diffs, tests, and implementation choices. Even when the coordinator did not accept a leaf's patch wholesale, it could inspect exact changes and decide what to land.

Proposal mode required a translation loop. The leaf translated implementation work into advice, and the coordinator translated that advice back into code. That loop can be useful for review, but it spends tokens on

prose and leaves the coordinator with more reconstruction work. In this experiment, that extra coordination showed up as higher Spark token use for proposal at every root reasoning level and higher total token use for proposal at low, medium, and high.

## Why GPT and Spark Tokens Increased

Spark tokens are expected to increase when Spark leaves are added: six xhigh leaves are doing additional implementation or review work. GPT tokens can also increase because the coordinator has more to read, reconcile, and integrate. Spark shifts the coordinator's job from pure implementation into planning, delegation, review, conflict resolution, and final integration.

That coordination cost matters most when comparing medium direct with high solo. Medium direct spends nearly as many GPT tokens as high solo (`2.436M` vs `2.612M`) before Spark tokens are counted. After Spark tokens are counted, medium direct is much more expensive in total implementation tokens (`3.843M` vs `2.612M`) while scoring lower quality.

## Why Solo High and Xhigh Were Strong Practical Choices

RuleLedger v3 stresses cross-language consistency more than independent bug fixing. Good solutions need one replay semantics model shared across TypeScript and Python, billing, reporting, summaries, parity outputs, and performance behavior.

Solo high and xhigh keep the full design in one implementation thread. Spark leaves add local search, but they also split context. The coordinator then has to merge partial solutions without introducing semantic drift. At medium, the extra search produced a large same-level lift. At xhigh, the best result came from preserving a single high-reasoning implementation thread.

## Limitations

- The benchmark is RuleLedger v3. Results may differ for tasks that are more modular, easier to partition, less parity-sensitive, or better suited to independent implementation shards.
- The solo comparison group has 50 runs per reasoning level; each Spark-assisted direct/proposal cell has 20 runs.
- The solo comparison group and Spark-assisted groups were collected in separate batches. A two-run GPT-only bridge was too noisy to support a drift adjustment.
- All Spark leaves used xhigh reasoning. Lower Spark reasoning levels and mixed Spark reasoning policies remain untested.
- The topology is fixed: one GPT-5.5 coordinator, six Spark leaves, and one GPT-5.5 integration pass. Deeper topologies, fewer leaves, stronger merge tooling, or different leaf assignments could change the result.
- Elapsed seconds are per-run harness measurements affected by batch parallelism and scheduler conditions. They should not be read as interactive latency measurements.
- Token counts are cost inputs, not prices. GPT and Spark tokens may have different pricing, quota limits, and opportunity costs.
- The efficiency recommendation uses ratio-of-means because it matches aggregate budget planning. Some narrow comparisons, including high solo versus medium direct GPT-token efficiency, are metric-sensitive under mean-of-ratios.

## Appendix A: Detailed Quality Table

| Root | Mode | Runs | Quality mean | Quality median | Quality sd | Hidden correctness | Hidden parity | Performance | Judge |
|---|---|---|---|---|---|---|---|---|---|
| low | solo | 50 | 0.434 | 0.447 | 0.181 | 0.364 | 0.531 | 0.556 | 0.394 |
| low | direct | 20 | 0.482 | 0.455 | 0.137 | 0.400 | 0.603 | 0.612 | 0.458 |
| low | proposal | 20 | 0.472 | 0.479 | 0.153 | 0.424 | 0.528 | 0.543 | 0.439 |
| medium | solo | 50 | 0.462 | 0.480 | 0.175 | 0.409 | 0.506 | 0.515 | 0.459 |
| medium | direct | 20 | 0.622 | 0.576 | 0.160 | 0.571 | 0.728 | 0.710 | 0.575 |
| medium | proposal | 20 | 0.529 | 0.489 | 0.114 | 0.439 | 0.661 | 0.663 | 0.522 |
| high | solo | 50 | 0.696 | 0.685 | 0.195 | 0.674 | 0.734 | 0.738 | 0.683 |
| high | direct | 20 | 0.712 | 0.691 | 0.158 | 0.680 | 0.775 | 0.775 | 0.698 |
| high | proposal | 20 | 0.652 | 0.593 | 0.162 | 0.635 | 0.736 | 0.718 | 0.575 |
| xhigh | solo | 50 | 0.755 | 0.709 | 0.171 | 0.739 | 0.791 | 0.800 | 0.752 |
| xhigh | direct | 20 | 0.734 | 0.693 | 0.152 | 0.720 | 0.750 | 0.750 | 0.751 |
| xhigh | proposal | 20 | 0.714 | 0.612 | 0.162 | 0.685 | 0.750 | 0.750 | 0.740 |

## Appendix B: Detailed Token Table

Efficiency columns use ratio-of-means.

| Root | Mode | Runs | GPT tokens | Spark tokens | Total tokens | Quality/GPT token | Quality/total token | Elapsed sec |
|---|---|---|---|---|---|---|---|---|
| low | solo | 50 | 815,190 | n/a | 815,190 | 5.32e-7 | 5.32e-7 | 199 |
| low | direct | 20 | 1,444,628 | 1,472,207 | 2,916,835 | 3.34e-7 | 1.65e-7 | 494 |
| low | proposal | 20 | 1,486,511 | 1,613,103 | 3,099,614 | 3.18e-7 | 1.52e-7 | 541 |
| medium | solo | 50 | 1,389,968 | n/a | 1,389,968 | 3.32e-7 | 3.32e-7 | 318 |
| medium | direct | 20 | 2,436,095 | 1,406,665 | 3,842,759 | 2.55e-7 | 1.62e-7 | 589 |
| medium | proposal | 20 | 2,508,418 | 1,811,092 | 4,319,510 | 2.11e-7 | 1.22e-7 | 628 |
| high | solo | 50 | 2,612,154 | n/a | 2,612,154 | 2.66e-7 | 2.66e-7 | 686 |
| high | direct | 20 | 2,801,135 | 632,108 | 3,433,243 | 2.54e-7 | 2.07e-7 | 650 |
| high | proposal | 20 | 2,933,084 | 1,563,753 | 4,496,837 | 2.22e-7 | 1.45e-7 | 751 |
| xhigh | solo | 50 | 3,333,886 | n/a | 3,333,886 | 2.26e-7 | 2.26e-7 | 1,019 |
| xhigh | direct | 20 | 3,877,968 | 610,527 | 4,488,495 | 1.89e-7 | 1.64e-7 | 951 |
| xhigh | proposal | 20 | 3,636,127 | 793,673 | 4,429,799 | 1.96e-7 | 1.61e-7 | 968 |

## Appendix C: Reproducibility

Key source files:

- Experiment plan: `plans/experiments/spark-mode-effic iency-direct-vs-proposal.md`
- Shared RuleLedger v3 prompt: `prompts/task_common_v3.md`
- Staged Spark prompt: `prompts/task_staged_spark_v3.md`
- Judge prompt: `prompts/judge.md`
- Scoring profile: `configs/scoring_v3.yaml`
- Analysis script: `scripts/analyze_spark_mode_efficiency.py`

Run directories preserve raw JSONL events, stderr logs, prompts, rendered configs, diffs, test logs, judge output, metadata, scores, and generated HTML and PDF reports.

Analysis command:

```
python scripts\analyze_spark_mode_efficiency.py `
  --experiment-dir runs\20260624T023048-spark_mode_efficiency_pilot-pilot `
  --experiment-dir runs\20260624T053702-spark_mode_efficiency_main-main `
  --experiment-dir runs\20260625T054133-spark_mode_efficiency_low_medium_extension-low_medium_extensi on_j7_j6 `
  --experiment-dir runs\20260624T220413-spark_mode_efficiency_high_extension-high_extension_j7_j6 `
  --experiment-dir runs\20260624T105944-spark_mode_efficiency_xhigh_extension-xhigh_extension `
  --output-dir runs\analysis\spark_mode_efficiency_final
```

Validation evidence:

- Final analysis ingested rows: `384`.
- Official comparison rows in this paper: `360`.
- All official run `validation.json` files: `passed`.
- Full repository test suite: `python -m pytest -q` (`193 passed in 11.96s`).