

Two Brains, One Pipeline

How to split the work between frontier models on claude.ai and coding agents for data processing and analytics projects.

A working guide: separation of duties, strategy and spec writing, dispatching coding agents, review discipline, and cost control. Prepared September 2026 by Seth Therrien, Onshore Consulting. (Transcribed from the operator's 14-page PDF on 2026-10-13; the original's "for a mentee" line is dropped, everything else is as written.)

Contents: 1 The core idea; 2 Separation of duties; 3 The working loop, end to end; 4 Walkthrough A: a nightly ingestion pipeline; 5 Walkthrough B: a metrics layer and an analytics dashboard; 6 Templates; 7 Pointers that save you weeks; 8 Good opportunities in data and analytics work; 9 Cost management; 10 Common failure modes; 11 A 30-day starter plan.

1. The core idea: thinking partner vs. execution engine

You have access to two very different kinds of AI help, and most of the value comes from not confusing them.

A frontier model in a chat interface (claude.ai) is a thinking partner. It is at its best when you bring it messy context and ask it to help you reason: what problem are we actually solving, what are the options, what are the risks, what should the design be, what does done look like, is this output correct. It has a long conversation memory, it can read the files you give it, it can search, and it can produce documents you will hand to other people or to other tools.

A coding agent (Claude Code, Cursor, Codex-style agents, and similar) is an execution engine. It lives in your repository, can read and edit many files, run tests, run scripts, inspect data samples, and iterate until something works. It is at its best when it has a precise, bounded task with a clear definition of done and a way to verify it.

The rule in one line: decide and specify in the chat; build and verify in the agent; review the result back in the chat. Never let the agent decide what to build, and never make the chat write the code you will paste by hand.

Why this split matters for data work specifically:

- Data projects fail on definitions far more than on code. What counts as an active customer, which timestamp is the event time, how to treat late-arriving rows. Those are thinking-partner questions.
- Data projects also produce a lot of boring, testable code: loaders, dedupe steps, schema checks, dbt models, dashboard queries. That is agent work, and agents are cheap and fast at it once the spec is clear.
- The expensive mistakes are silent ones: a join that fans out, a filter that drops a segment, a metric that quietly changed meaning. A review step with a second model, working

from the spec, catches a surprising number of these.

2. Separation of duties

Use this table as the default assignment. Move a task across only when you have a specific reason.

Activity	Frontier model in claude.ai	Coding agent in the repo
---	---	---
Understanding the problem	Yes. Bring source docs, sample rows, stakeholder notes. Ask it to restate the problem and list unknowns.	No.
Strategy and options	Yes. Compare approaches, name trade-offs, recommend one with reasons.	No.
Data modeling and definitions	Yes. Grain, keys, metric definitions, edge cases, naming.	No. It implements the model you chose.
Writing the spec	Yes. Produce the spec document, iterate until it is unambiguous.	No.
Writing the dispatch prompt	Yes. Turn the spec into a bounded task with acceptance criteria.	No.
Writing code	Rarely. Only tiny snippets to illustrate a design.	Yes. All real code, tests, migrations, SQL.
Running code and tests	No (unless you use its sandbox for throwaway analysis).	Yes. It runs, reads errors, fixes, reruns.
Profiling a dataset	Sometimes, for a small sample you upload.	Yes, against real files or a dev database.
Debugging	Second opinion on a confusing error or a suspicious result.	First line. It has the stack trace and the files.
Reviewing the result	Yes. Diff plus spec plus test output. Ask what could be wrong.	Can self-review, but never the only reviewer.
Writing docs and runbooks	Yes, from the agent's summary and your notes.	Inline docstrings, README stubs.
Deciding what is next	Yes, with you.	No.

Three reasons the split is drawn this way:

1. Context shape. The chat model sees the whole story across a long conversation; the agent sees the repository and the terminal. Put decisions where the story is, and implementation where the files are.
2. Verifiability. Agents can run things. A test that passes is evidence. A paragraph that sounds right is not.
3. Cost and blast radius. Agent runs consume tokens per file read and per retry. A vague prompt makes an agent wander, which is expensive and produces sprawl you then have to clean up.

3. The working loop, end to end

Every meaningful piece of work goes through the same seven stages. Small tasks compress the stages into minutes; large ones take days. The stages do not change.

Stage	Where	Output
-------	-------	--------

| --- | --- | --- |

- | 1. Frame | claude.ai | One paragraph: the problem, who cares, what changes if it is solved. |
- | 2. Discover | claude.ai plus your own digging | List of sources, sample data, constraints, known unknowns. |
- | 3. Strategy | claude.ai | Options considered, one chosen, why. Risks and what would change the choice. |
- | 4. Spec | claude.ai | A document a stranger could implement from. Grain, schema, rules, acceptance tests. |
- | 5. Dispatch | claude.ai writes it; you paste it | One bounded task prompt with definition of done. |
- | 6. Execute | coding agent | Code, tests passing, a short summary of what changed and what it noticed. |
- | 7. Review and log | claude.ai | Findings, fixes needed, decision log entry, next dispatch. |

How to run the loop in practice:

1. Open a dedicated project or conversation in claude.ai per initiative. Attach the source docs, a data dictionary if one exists, and a small sample of real rows (with anything sensitive removed).
2. Set persistent instructions in the project: your stack, naming conventions, the definition-of-done you expect, and any rules (for example: never invent column names; ask when a definition is ambiguous).
3. Work stages 1 through 4 in that conversation. Push back on the model. Ask it to list the ways the plan could be wrong. Ask it what it would need to know to be confident.
4. Have it write the dispatch prompt as a single block you can paste. Paste it into the agent unchanged.
5. Let the agent run. Do not micromanage it mid-task; if it asks a question you did not anticipate, that is a hole in the spec. Answer it, then fix the spec.
6. Bring the agent's summary, the diff (or the PR link), and the test output back to the chat. Ask for a review against the spec. Fix, redispach, or accept.
7. Log the decision and the outcome in one place (a decisions file in the repo, a board, a wiki page). Future you will need to know why the metric is defined the way it is.

Spec-first, config-over-code. When a rule can be expressed as configuration (a YAML mapping of source columns to target columns, a list of dedupe keys, a table of metric definitions), make the spec produce that configuration and make the agent write generic code that reads it. You get fewer bugs, easier review, and reuse across projects.

4. Walkthrough A: nightly ingestion from raw files to a warehouse table

Situation. A small company receives daily CSV exports from a vendor into cloud storage. They want a clean, deduplicated orders table in their warehouse, refreshed nightly, with a way to know when the load failed or the file looked wrong.

Stage 1 and 2: frame and discover (claude.ai). You upload two sample files (a normal day and a weird day), the vendor's export documentation, and a note from the ops lead about what they need. You ask:

I am building a nightly load of vendor order exports into our warehouse.

Attached: two sample CSVs, the vendor doc, and the ops lead's note.

Before proposing anything, please:

1. Restate the goal in your own words.
2. List everything you can infer about the data (grain, keys, types, quirks).
3. List every question you would need answered before writing a spec.

Do not propose a design yet.

A good answer will surface things like: order_id repeats across days when an order is updated; the timestamp column has two formats; a header row appears twice on one file; amounts are strings with currency symbols. Answer the questions you can, go find the rest, and come back.

Stage 3: strategy.

Here are the answers. Now give me two or three viable designs for the load, with trade-offs. I care about: simplicity to operate, ability to reprocess a bad day, and catching a malformed file before it lands in the clean table. Recommend one and tell me what would change your recommendation.

Typical outcome: land raw files as-is into a raw table partitioned by file date; a validation step that checks row count, required columns, and type coercion and quarantines the file on failure; a merge step keyed on order_id taking the latest updated_at; a small run log table. Reprocessing a day means deleting that partition and rerunning.

Stage 4: spec. Ask the chat to write the spec using the template in section 6. Insist on these sections being concrete:

- Grain: one row per order_id in the clean table; one row per (file_date, source_row_number) in raw.
- Column mapping: a table of source column, target column, type, transformation, null handling. This becomes a YAML file the loader reads.
- Dedupe rule: keep the row with the greatest updated_at; ties broken by the later file_date.
- Validation rules and what happens on failure: quarantine, alert, do not touch the clean table.
- Acceptance tests: given the two sample files, the clean table has N rows, order X shows the updated amount, the malformed file lands in quarantine and the run log records the reason.
- Out of scope: anything the agent might be tempted to build (dashboards, a scheduler UI, retries).

Stage 5: dispatch prompt (written by the chat, pasted into the agent).

LOAD-01: Vendor orders nightly load

Context: read docs/specs/vendor-orders-load.md fully before doing anything. The mapping lives in config/vendor_orders_mapping.yaml (create it from the spec's column table). Do not hardcode column names in Python.

Build:

- src/loads/vendor_orders.py with functions: land_raw(file_path), validate(file_path) -> ValidationResult, merge_clean(file_date).
- A CLI entry point: python -m loads.vendor_orders --date YYYY-MM-DD
- Tests in tests/test_vendor_orders.py using the two fixtures in tests/fixtures/ (copied from the spec's sample files).

Definition of done:

- All acceptance tests in spec section 7 pass. Show me the test output.
- Running the CLI twice for the same date is idempotent (same row count).
- No new dependencies beyond what is in pyproject.toml without asking.

When finished, write a short summary: files changed, anything in the spec that was ambiguous, and anything you noticed in the data that the spec did not anticipate. Do not start on scheduling or alerting.

Stage 6 and 7: execute, then review. The agent builds it and reports. You paste its summary and the diff back into the chat:

Here is the agent's summary and the diff for LOAD-01. Review against the spec.

1. Does the dedupe rule match section 4.3 exactly?
2. Is there any path where a failed validation still writes to the clean table?
3. What is not tested that should be?
4. Anything you would send back?

Common catches at this stage: the tiebreak on dedupe was implemented backwards; validation checked column presence but not order; the idempotency test was passing only because the fixture had no duplicates. Each becomes a one-line redispatch. Then log the decision (why latest-updated_at wins) and move on.

5. Walkthrough B: a metrics layer and an analytics dashboard

Situation. Leadership asks for a weekly dashboard: active customers, revenue, churn, and a few funnel numbers. The data is in a warehouse already, spread across five tables built at different times by different people. Two of the existing reports disagree with each other.

Where people go wrong. The instinct is to open the agent and say build a dashboard. The agent will happily build one. It will pick a definition of active customer, it will pick a join path, and it will produce charts that look plausible and are wrong in ways nobody can see. Three weeks later someone asks why churn dropped in March and nobody can answer.

The right order:

1. Definitions first, in the chat. Upload the schema of the five tables and the two disagreeing reports. Ask the model to reconcile them: where do they diverge, and what definition would each side have to accept. Bring the reconciled definitions to the stakeholders. Get sign-off in writing.
2. Metric spec. One table: metric name, grain, formula in words, formula in SQL-ish pseudocode, source tables, filters, known caveats, owner. This is the contract. It goes in the repo.

3. Model the layer. Ask the chat for a layered design: staging models that clean each source, an intermediate customer-day or customer-week table, and a small set of mart tables the dashboard reads. Decide naming and materialization once.
4. Dispatch per layer. One dispatch for staging (five models plus schema tests), one for the intermediate table, one for marts, one for the dashboard. Each with its own acceptance checks: row counts reconcile to a known number, a hand-computed metric for one week matches the mart to the cent.
5. Review by reconciliation. Before the dashboard ships, ask the chat to design three reconciliation queries (for example: mart revenue equals sum of invoice amounts for the same period). Dispatch them as tests. If they fail, the layer is wrong, not the dashboard.

Sample metric spec row:

Field	Value
---	---
Metric	active_customers_weekly
Grain	One value per ISO week
Definition (words)	Count of distinct customers with at least one paid order in the week, excluding refunded-in-full orders and internal test accounts.
Pseudocode	COUNT(DISTINCT customer_id) WHERE order_status = 'paid' AND refund_amount < order_amount AND is_internal = false GROUP BY iso_week
Sources	stg_orders, stg_refunds, dim_customer
Caveats	Refunds can arrive up to 30 days later; last 4 weeks are provisional.
Owner	Finance lead (sign-off 2026-09-18)

Dispatch prompt shape for the mart layer:

MART-01: Weekly customer metrics mart

Read docs/specs/metrics.md (the metric table) and models/intermediate/customer_week.sql before starting. Build models/marts/weekly_metrics.sql implementing every metric in the table exactly as specified. Add schema tests: not_null on iso_week, unique on iso_week, accepted range on churn between 0 and 1.

Acceptance: for iso_week 2026-W30, active_customers_weekly must equal 1,842 and revenue_weekly must equal 213,904.17 (hand-verified). Include a test that asserts both. Report any metric whose pseudocode cannot be implemented as written, and stop rather than improvise.

Notice the hand-verified number. One known-good value for one period, computed by you or the stakeholder by an independent path, is the single highest-leverage test you can add to any analytics project. Ask the chat to help you pick which number and how to compute it independently.

6. Templates

6.1 Strategy brief (one page, written in claude.ai)

<Initiative name> -- Strategy brief

Problem: <one paragraph, plain language, who is affected>
Success looks like: <2-3 observable outcomes>
Constraints: <stack, budget, deadline, compliance, people>
Options considered:
A. <name> -- pros / cons / rough effort
B. <name> -- pros / cons / rough effort
C. <name> -- pros / cons / rough effort
Recommendation: <one option> because <reasons>
What would change this: <the facts that would flip the choice>
Risks and mitigations: <top 3>
Open questions: <numbered, with who can answer>

6.2 Spec (the document the agent implements from)

<Component name> -- Spec vX.Y

1. Purpose and scope (and explicit out-of-scope list)
2. Inputs: sources, formats, arrival pattern, sample files referenced by path
3. Outputs: tables/files, grain, primary keys, schema table (name, type, nullable, description)
4. Rules: transformations, dedupe, late data, null handling, time zones, currency, rounding. Number each rule (4.1, 4.2 ...) so reviews can cite them.
5. Configuration: which rules live in config files and their shape
6. Failure behavior: what happens on bad input, partial success, reruns
7. Acceptance tests: concrete, numbered, each with input and expected output
8. Operational notes: how to run, how to reprocess, what to monitor
9. Decisions log: date, decision, why, who

6.3 Dispatch prompt

<TASK-ID>: <short title>

Read first: <spec path(s)>, <relevant existing files>
Build: <files and functions/models, named>
Do not: <the two or three things it will be tempted to do>
Definition of done:
- <acceptance tests by number> pass; show output
- <idempotency / lint / type checks as applicable>
- No new dependencies without asking
Report back: files changed; ambiguities found; surprises in the data;
what you would do next (but do not do it).

6.4 Review checklist (paste into claude.ai with the diff and summary)

1. Does every numbered rule in the spec have a corresponding line of code or config? Which are missing?
2. Does every acceptance test actually exercise the rule it claims to? Could it pass for the wrong reason?
3. Where does data enter and leave? Is there any path from a failed check to a written output?
4. Any join that could change row count? Any filter that could silently drop a segment?

5. Any hardcoded value that belongs in config? Any config the code ignores?
6. What would you want to know before running this on a full day of real data?
7. One-line verdict: accept, accept with follow-ups, or send back with a numbered list.

7. Pointers that save you weeks

- Give the chat real samples, not descriptions. Twenty real rows (scrubbed) beat two paragraphs about the data.
- Ask for questions before answers. The first prompt on any new problem should end with: list what you would need to know. It converts hidden assumptions into visible ones.
- Number everything. Rules, tests, options, findings. You will refer back to them by number in dispatches and reviews, and the agent will too.
- One dispatch, one concern. If the prompt has the word and joining two unrelated deliverables, split it.
- Stop conditions beat instructions. "Stop and report if the spec cannot be implemented as written" prevents the agent from inventing behavior.
- Keep the spec in the repo, not the chat. The chat produces it; the file is the source of truth; the agent reads the file.
- Use a fresh agent session per dispatch. Long agent sessions accumulate stale context and cost more per action. Fresh session plus a good prompt is faster and cheaper.
- Use projects in claude.ai per initiative. Persistent instructions and attached docs mean you stop re-explaining your stack every conversation.
- Use the chat's sandbox for throwaway analysis. Uploading a CSV and asking for a quick profile or a chart is faster than dispatching an agent for a one-off look.
- Make the agent show its evidence. Test output, row counts, a sample of the result. A claim without output is not done.
- Write the decisions log as you go. Ten seconds now saves an hour of archaeology later.
- Never paste secrets into either tool. Reference credentials by alias and where they live (for example an environment variable name or a secrets manager path). The agent reads them at runtime; the chat never needs the value.

8. Good opportunities in data and analytics work

These are the kinds of tasks where the split pays off fastest. They are repetitive enough for an agent to execute reliably, and valuable enough that a good spec matters.

Opportunity	Chat does	Agent does
---	---	---
Source onboarding	Profiles samples, drafts the mapping, flags quirks	Builds loader from the mapping config, writes tests
Data quality suite	Chooses which checks matter and thresholds	Implements checks, wires alerts, adds to CI
Metric reconciliation	Designs independent reconciliation paths	Writes the queries as tests, reports diffs
Schema evolution	Decides the migration strategy and backfill rules	Writes migration, backfill script, and rollback
Legacy SQL cleanup	Reads a gnarly query, restates intent, proposes a layered rewrite	Rewrites into staging and marts, proves equivalence with a diff test
Documentation	Turns agent summaries and specs into readable docs	Generates docstrings, column descriptions, lineage stubs

| Ad hoc analysis to product | Works through the analysis in the sandbox, decides what to productionize | Turns the notebook logic into a tested module and scheduled job |
| Cost and performance | Reads query plans and cost reports, recommends targets | Adds partitioning, clustering, incremental logic; measures before and after |

Two patterns worth building once and reusing:

1. A generic config-driven loader. One Python module that takes a mapping file and produces raw, validated, and merged tables. Every new source becomes a spec plus a YAML file, not a new codebase.
2. A reconciliation harness. A small framework where each test is a pair of queries expected to match within a tolerance. Every new metric ships with at least one.

9. Cost management

Costs show up in three places: your chat subscription or API usage, the agent's token consumption, and the compute your pipelines burn. The habits below keep all three in check.

Chat-side (claude.ai)

- Use the most capable model for strategy, specs, and reviews. The cost of a wrong definition dwarfs the cost of a strong model for an hour.
- Start a new conversation when the topic changes. Very long threads carry every earlier message into every new reply.
- Attach documents to a project once instead of pasting them repeatedly.
- Ask for the deliverable (the spec, the prompt) rather than a discussion about the deliverable when you already know what you want.

Agent-side

- The spec is the cost control. A precise dispatch finishes in one pass; a vague one triggers exploration, retries, and rework.
- Point the agent at exactly the files it needs. Reading a whole repository is expensive and rarely necessary.
- Keep fixtures small. Ten representative rows, not ten thousand. Full-data runs happen in a scheduled job, not inside the agent.
- Set explicit stop conditions and a scope fence. Every unrequested feature costs tokens twice: once to build and once to remove.
- Prefer a cheaper or faster model for mechanical tasks (renames, boilerplate tests, formatting) and the strongest model for tasks with real design risk. Many agent tools let you switch per task.
- Run linters, type checks, and formatters as pre-commit hooks so the agent does not spend tokens on style feedback.
- Watch for loops. If an agent is on its third attempt at the same failing test, stop it, bring the failure to the chat, and fix the spec or the approach.

Pipeline and warehouse compute

- Ask the chat for a cost review of any query or model before it is scheduled: partition pruning, incremental strategy, and whether the refresh cadence matches how often anyone

looks.

- Build incrementally by default. Full rebuilds are for backfills and are dispatched as their own explicit task.
- Add a run log table with duration and bytes scanned. Review it weekly with the chat; ask it to spot the three most expensive steps and propose fixes.
- Keep development against a sampled dataset or a dev schema, never against production volumes.

Rule of thumb. Spend generously on thinking (strategy, spec, review) and frugally on execution (bounded prompts, small fixtures, right-sized models). This is the opposite of the instinct most people start with, and it is the single biggest lever on both quality and cost.

10. Common failure modes and how to avoid them

| Failure mode | What it looks like | Fix |

| --- | --- | --- |

| Agent as architect | You describe a goal to the agent and it picks the design. The design is plausible and unreviewed. | Decide in the chat. Dispatch only after the spec exists. |

| Chat as coder | You paste code from the chat into files by hand, fix indentation, lose track of what changed. | The chat writes prompts; the agent writes code. |

| Definition drift | Two reports disagree because active means different things in each. |

Metric spec with owner sign-off, stored in the repo, referenced by tests. |

| Silent join fan-out | Row counts balloon; totals inflate; nobody notices for weeks. |

Reconciliation tests on counts and sums; review question 4 in the checklist. |

| Scope creep in dispatch | You asked for a loader and got a loader, a scheduler, alerting, and a dashboard. | Explicit out-of-scope list and a stop instruction in every prompt. |

| Untested acceptance | Tests pass because fixtures never exercise the hard case. | Include the weird-day sample; have the chat check that tests fail without the fix. |

| Lost decisions | Six months later nobody knows why refunds are excluded. | Decisions log in the spec, one line per decision, updated at review time. |

| Secrets in prompts | A connection string ends up in a chat or a prompt file. | Alias plus storage path only. Environment variables at runtime. |

11. A 30-day starter plan

Pick one real but modest project. A single source onboarding or a small metrics layer is ideal.

Week 1: set up the loop

1. Create a project in claude.ai for the initiative. Add persistent instructions: stack, conventions, definition-of-done rules, the requirement to ask before inventing.
2. Create the repo structure: docs/specs, config, src, tests/fixtures, docs/decisions.md.
3. Run stage 1 and 2 (frame and discover) fully. Produce the strategy brief.

Week 2: first spec and first dispatch

1. Write the spec in the chat using the template. Iterate until the chat can no longer find an ambiguity.

2. Dispatch one small task. Review it with the checklist. Note what the spec missed; fix the spec before the next dispatch.

Week 3: build out

1. Dispatch the remaining layers one at a time. Add the reconciliation harness as its own dispatch.
2. Start the weekly cost review habit: run log, top three expensive steps, one fix.

Week 4: reflect and standardize

1. Ask the chat to review the whole decisions log and the spec history. What would you change in the templates?
2. Turn the generic pieces (loader, reconciliation harness, prompt templates) into reusable assets.
3. Write a one-page retrospective: what the split saved you, what it cost, what you will do differently.

Final thought. The skill you are building is not prompting. It is the discipline of deciding clearly before building, and verifying before believing. The tools make that discipline cheap to practice. Practice it on every task, including the small ones, and it becomes the way you work.