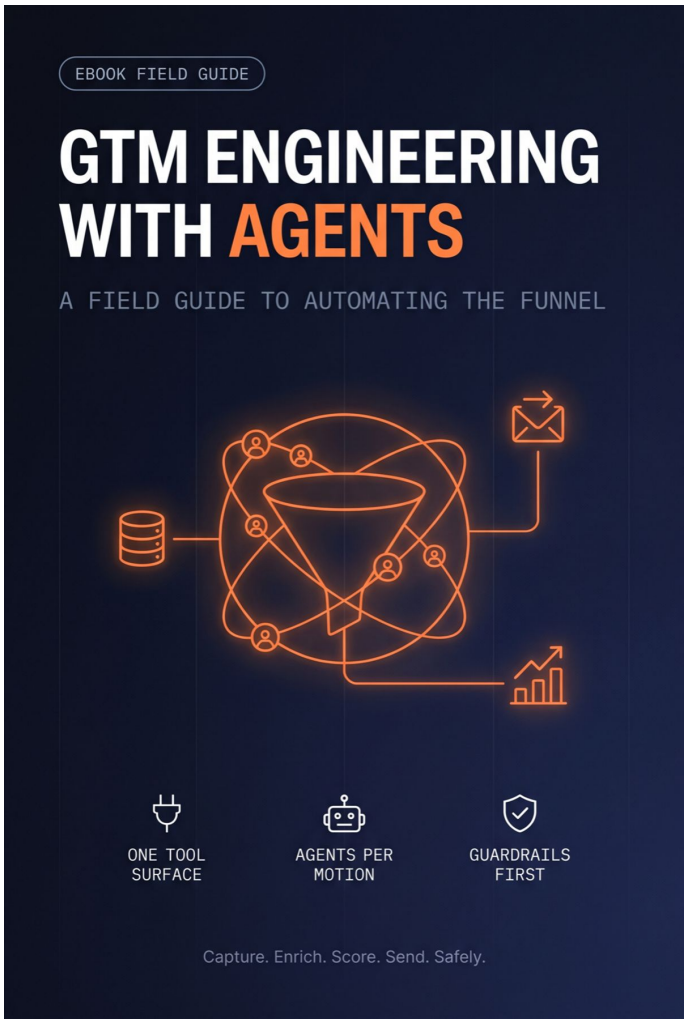

GTM Engineering with Agents: A Field Guide to Automating the Funnel

Every go-to-market team is really running a handful of pipelines. Leads come in, get cleaned up, get scored, get worked, and either turn into revenue or quietly rot. Most of that work is mechanical, and most of it gets done by hand or by a tangle of point-to-point integrations that break the moment a vendor changes an endpoint.

This guide takes the agentic approach the rest of the series is built on and points it straight at the funnel. The idea is simple: wrap every source you care about (your CRM, your enrichment providers, your email, your product analytics) as an MCP tool, then let small agents orchestrate the motions on top of them. Capture, enrichment, scoring, outbound, nurture, hygiene, forecasting,

churn signals. One tool surface, many agents, each running on a schedule.

It's hands-on. We build each motion as a small agent loop over MCP tools, wire them together with a cron, and put guardrails around the parts that can email a customer or overwrite a record. By the end you'll have a mental model for GTM as a system of agents, plus enough concrete patterns to start replacing the manual busywork on your own stack.

It pairs with [MCP from Scratch](/guides/mcp-from-scratch) for the protocol and [Running the Fleet](/guides/running-the-fleet) for orchestration. This one is about pointing all of that at pipeline.

This is a living document and will be updated as the tools and patterns evolve.

Roger Stringer · rogerstringer.com

August 14, 2026

Contents

The Funnel Is Already a Pipeline

I spent a week a few years back sitting next to a two-person sales team at a company I was consulting for, mostly so I'd know what to build them. What I found was a pipeline nobody had bothered to write down.

A form fills. Someone opens the CRM to check whether that person is already in there. Someone opens a browser tab and looks up the company. Someone decides, based on a rule that lives entirely in their head, whether this is worth an hour. Someone writes an email. Four days later someone remembers to follow up. At the end of the quarter, someone spends two days deleting duplicates and fixing the deals that have been sitting in "negotiation" since March.

Every one of those steps has an input, an output, and a rule. That's a pipeline. The reason it doesn't feel like one is that the steps live in different heads, different tabs, and a pile of point-to-point integrations that each know about exactly two systems and nothing about the funnel.

There's the fragility. Your form-to-CRM zap knows the form and the CRM. It doesn't know that the same person filled out a different form last month under a different email, or that their company just tripled its seat count, or that a rep already has a thread going. Nothing in the system sees the whole shape, so the whole shape lives in people, and people are expensive and they go on vacation.

What we're building instead

The move in this guide is boring and it works: put one tool surface over every system you care about, then run small agents on top of it.

Your CRM, your enrichment providers, your email, your product analytics, your billing. Each one gets wrapped as MCP tools. Then each motion in your funnel gets an agent: one that cleans up inbound leads, one that enriches, one that scores, one that drafts outbound, one that watches for churn signals, one that runs the janitorial pass at 6am. Each is small. Each runs on a schedule. Each can see everything, because they all share the same tool surface.

I want to be blunt about what this is not, because I've watched people try it. This is not an "AI SDR" you point at your funnel and walk away from. What that gets you is a very confident machine sending very bad email to your best account. What we're building is closer to a set of scheduled jobs that happen to read, reason, and write in plain English, with hard code wrapped around every part that can touch a customer.

What you should already have

This guide assumes the protocol layer from MCP from Scratch (/guides/mcp-from-scratch) and the orchestration patterns from Running the Fleet (/guides/running-the-fleet). You don't have to have read either one, but I'm not going to re-teach how to stand up an MCP server or how a scheduler routes work between agents. I'll link back where it matters.

What you do need is a funnel with real data in it and enough frustration to want it automated.

Start with the surface. Everything else sits on top of it.

One Tool Surface, Many Agents

Here's the inversion that makes everything else possible.

A normal GTM stack is a graph of integrations. Form to CRM. CRM to enrichment. Enrichment back to CRM. CRM to email. Email to CRM. Each edge is a separate piece of glue with its own auth, its own field mapping, and its own way of failing silently. Add a system and you add edges to everything it touches.

Wrap each system as an MCP server instead and the graph collapses into a star. Every agent talks to one surface. Adding a system adds one server, not fifteen edges.

Name tools for the funnel, not the vendor

This is the part people get wrong. It's tempting to expose your CRM's API one-to-one, because that's the least work. Don't. Name the tools after the job:

```
// crm-server: what the funnel needs, not what the vendor ships
server.tool(
  "find_person",
  "Look up a person by email, or by name plus company domain.",
  {
    email: z.string().email().optional(),
    name: z.string().optional(),
    domain: z.string().optional(),
  },
  async ({ email, name, domain }) => { /* ... */ }
);

server.tool(
  "list_stale_deals",
  "Deals with no activity in N days, optionally filtered by stage.",
  { days: z.number().default(30), stage: z.string().optional() },
  async ({ days, stage }) => { /* ... */ }
);
```

Your CRM doesn't offer `list_stale_deals`. It's three API calls and a date filter, and it's exactly the question your hygiene agent asks every night. Put the assembly in the server. The agent should spend its tokens on judgment, not on figuring out pagination.

The payoff shows up the day you switch enrichment providers. You rewrite one server. Every agent that scores, routes, and drafts keeps working, because none of them ever knew the vendor's name.

Split reads from writes

Run two servers per system, or at minimum two tool groups with separate credentials: one read-only, one write.

```
crm-read     find_person, list_deals, list_stale_deals, get_activity
crm-write    upsert_person, update_deal_stage, add_note
mail-read    list_threads, get_thread
mail-send    send_message
```

Now "give this agent access to the CRM" is a real decision with two possible answers. Your forecasting agent gets `crm-read` and nothing else, and no prompt injection hiding in a lead's job title field can make it overwrite a record, because the capability isn't in the room. Your outbound drafter gets `crm-read` and `mail-read` but never `mail-send`; sending happens in a separate, dumber process after a human approves.

I'd rather have ten narrow servers than one wide one. The narrowness is the security model.

One agent per motion

With the surface in place, each funnel motion becomes a small agent with a system prompt, a tool allowlist, and a schedule:

Agent	Tools	Runs
capture	`crm-read`, `crm-write`	every 15 min
enricher	`crm-read`, `crm-write`, `enrich-read`	hourly
scorer	`crm-read`, `crm-write`	hourly
drafter	`crm-read`, `mail-read`, `queue-write`	7am daily
janitor	`crm-read`, `report-write`	6am daily
signals	`crm-read`, `analytics-read`, `report-write`	daily
forecaster	`crm-read`, `report-write`	Monday 8am

Seven small agents beat one big one for the same reason seven small functions beat one big one. You can read them, test them, and turn one off without turning off the rest.

Before any of them can cooperate, though, they need to agree on what a "person" is. That's next.

The Record Everything Agrees On

Five agents writing to the same CRM without an agreed shape is how you end up with `title`, `job_title`, and `role` all populated, all slightly different, all written by something you built.

Before the first agent runs, define the record. Not the CRM's schema, which you don't control, but the shape your agents read and write through:

```
type Person = {
  id: string;
  email: string;
  name: string | null;
  title: string | null;
  accountId: string | null;
  stage: "new" | "qualified" | "working" | "won" | "lost" | "dead";
  score: number | null;
  fields: Record<string, FieldValue>;
};

type FieldValue = {
  value: string | number | null;
  source: string;      // "webform" | "provider:acmedata" | "agent:enricher"
  confidence: number;  // 0..1
  updatedAt: string;   // ISO
};
```

The `FieldValue` wrapper is the whole idea. Every value an agent writes carries where it came from and how sure it was.

Why provenance is not optional

Three reasons, and I've been burned by all of them.

First, precedence. When the enricher says the title is "VP Engineering" and the person typed "engineering leader" into your form, you need a rule, and the rule needs data to work on. Human-entered beats provider, provider beats agent-inferred. That's one line of code if you have `source`, and an argument in Slack if you don't.

Second, rollback. Six weeks in you'll discover the scorer has been reading a field the enricher was filling in wrong. With provenance you write `where source = 'agent:enricher' and updatedAt > '...'` and undo exactly that. Without it you're restoring a backup and losing everyone's real work along with it.

Third, trust. When a rep asks why a lead got a 40, you want to point at the values that produced it and say where each one came from. "The model decided" ends the conversation badly.

Signals are separate from fields

Fields describe what someone is. Signals describe what happened. Keep them apart:

```
type Signal = {
  accountId: string;
  kind: "usage_drop" | "seat_limit" | "champion_inactive" | "pricing_page_visit";
  value: number;
  observedAt: string;
  detail: string;
};
```

Signals are append-only. Nothing overwrites a signal, because it's a fact about a moment. Your churn agent, your expansion agent, and your forecaster all read the same signal stream and reach their own conclusions

from it, which is much healthier than three agents each computing "is this account in trouble" from raw analytics and getting three answers.

Write the mapping once

Your CRM will have its own field names, and they'll be ugly. Map them in the server, in one file, and never again:

```
const CRM_FIELDS = {
  title: "job_title__c",
  score: "lead_score_v2__c",
  stage: "status",
} as const;
```

When someone in ops renames a field (they will), you fix one line and every agent keeps working.

With a shape agreed, we can start filling it. The first agent goes at the front door.

Capture: Clean at the Door

Every duplicate in your CRM got in through the front door. Cleaning up later is a worse job than not letting it happen, so the first agent goes at ingest.

Here's the trap though: most of capture is not a job for a model.

Rules first, model second

Deduplication is mostly string comparison, and a model doing string comparison is slow, expensive, and occasionally creative. Do the deterministic work in code:

```
async function matchPerson(input: InboundLead) {
  // 1. exact email, case-normalized
  const byEmail = await crm.findPerson({ email: norm(input.email) });
  if (byEmail) return { match: byEmail, confidence: 1, via: "email" };

  // 2. same name at the same corporate domain
  const domain = domainOf(input.email);
  if (domain && !isFreeMail(domain)) {
    const candidates = await crm.findPerson({ name: input.name, domain });
    if (candidates.length === 1) {
      return { match: candidates[0], confidence: 0.9, via: "name+domain" };
    }
  }

  return { match: null, confidence: 0, via: "none" };
}
```

That handles the large majority. What's left is the genuinely ambiguous middle, and that's what the agent is for:

- Is `acme.com` the same account as `acme-corp.io`?
- Is "Sam Rivera" at Acme the same person as the "S. Rivera" record with a personal address and Acme in the notes?
- Did this company get acquired, and is the new domain on the record actually the parent?

Those need a look around. Give the capture agent `crm-read` plus a web search tool, ask for a decision with reasoning and a confidence number, and route anything under your threshold to a human queue instead of guessing.

Normalize before you store

Half of dedupe is preventing the problem:

```
const clean = {
  email: input.email.trim().toLowerCase(),
  name: titleCase(collapseSpaces(input.name)),
  domain: stripWww(domainOf(input.email)),
  company: dropSuffix(input.company), // "Acme Corp LLC" -> "Acme Corp"
  phone: toE164(input.phone, input.country),
};
```

None of that needs intelligence. All of it needs doing.

What the capture agent actually writes

Keep the write narrow. Capture creates or merges a person, sets `stage: "new"`, records the source, and stops. It does not enrich, score, assign an owner, or email anybody. Those are other agents' jobs, running on

their own schedules with their own failure modes.

I run capture every fifteen minutes rather than on a webhook, on purpose. Batching lets it see five leads from the same company arriving in the same hour and treat them as one account event, and a fifteen-minute delay has never once cost me a deal.

Try this

1. Export the last 500 leads that entered your CRM.
2. Run the rules-only matcher over them and count how many would have been caught as duplicates at the door.
3. Look at the ones the rules missed. That set is your capture agent's actual job description, and it's usually smaller and weirder than you expected.

New leads land clean. Most of them are still nearly empty. Filling them in is where the money goes, so that's next.

Enrichment Without Burning the Budget

Enrichment is where GTM automation quietly gets expensive. Providers charge per lookup, agents charge per token, and the default instinct (enrich everything the moment it arrives) means you're paying to research people who will never be worked.

Two rules keep the bill sane. Enrich late, and enrich in a cascade.

Enrich late

Don't enrich at capture. Enrich at the moment something needs the data, which in practice means right before scoring, and only for records that clear a cheap filter.

```
const worthEnriching = (p: Person) =>
  !isFreeMail(domainOf(p.email)) &&
  !isCompetitor(p) &&
  p.stage === "new";
```

On the funnels I've worked on, that filter alone cuts enrichment volume by half or more. Free-mail signups and competitor tire-kickers are a big share of raw inbound, and none of them need a firmographics lookup.

Cascade from cheap to expensive

Stack your sources by cost and stop at the first one that answers:

```
async function enrich(person: Person) {
  const steps = [
    () => cache.get(person.domain),        // free
    () => internal.lookupAccount(person),  // free, your own data
    () => providerA.person(person.email),  // cheap
    () => providerB.company(person.domain),// expensive
    () => agentWebLookup(person),          // slow, last resort
  ];

  const out: Partial<Fields> = {};
  for (const step of steps) {
    const result = await step().catch(() => null);
    merge(out, result);
    if (isComplete(out)) break;
  }
  return out;
}
```

`isComplete` is doing real work there. Define what "enough" means as a field list, per segment if you need to. Without it the cascade runs to the bottom every time and you've built an expensive way to call your expensive provider.

Cache by domain, not by person. Company data (industry, size, funding, tech stack) is identical for everyone at Acme, and it's the part providers charge most for. A domain-level cache with a 30 day TTL is the single highest-leverage thing in this chapter.

Where the agent earns its keep

Providers return facts. They're bad at questions like "does this company actually look like our customers?" That's the agent's part of the job, and it runs on the enriched record rather than on the raw web:

> Given this company profile and these ten examples of our closed-won accounts, describe in two

> sentences how this company is similar and how it differs. Name the single biggest reason they might not be a fit.

That paragraph is worth more than another twelve firmographic fields, and it feeds straight into the next agent.

Write it back with provenance

Every enriched field goes in wrapped, per the shape from the last chapter:

```
await crm.upsertPerson(person.id, {
  title: { value: out.title, source: "provider:a", confidence: 0.8, updatedAt: now() },
  employees: { value: out.employees, source: "provider:b", confidence: 0.9, updatedAt:
now() },
});
```

Never let enrichment overwrite a human-entered value. That rule lives inside `upsertPerson`, in code, once, where no prompt can talk it out of it.

Now the record is full. Time to decide whether anyone should care about it.

Scoring You Can Argue With

Lead scoring has a credibility problem, and it predates AI. Every rep I've worked with has, at some point, been handed a number by a system and quietly ignored it, because the number showed up without an argument attached.

An agent that scores is only useful if it shows its work.

Put the rubric in a file

Not in a prompt string buried in code. In a file, in version control, that a non-engineer can read and edit:

```yaml
# rubric.yml
fit:
  - id: fit.size
    weight: 20
    test: "Company has 50 to 2000 employees"
  - id: fit.stack
    weight: 15
    test: "Uses Postgres or MySQL somewhere in their stack"
  - id: fit.role
    weight: 15
    test: "Contact's title suggests budget authority for tooling"
intent:
  - id: intent.pricing
    weight: 20
    test: "Visited the pricing page in the last 14 days"
  - id: intent.multi
    weight: 15
    test: "Two or more people from this domain signed up this month"
disqualify:
  - id: dq.competitor
    test: "Company is a direct competitor"
  - id: dq.student
    test: "Free-mail address with no company context"
```

The agent reads the rubric, reads the record, and returns which lines fired:

```
{
  "score": 65,
  "fired": ["fit.size", "fit.stack", "intent.pricing"],
  "missed": ["fit.role", "intent.multi"],
  "disqualified": null,
  "note": "Ops lead at a 300-person logistics company, hit pricing twice last week. Title
suggests influence rather than budget, so pair with a director-level contact before
pushing a quote."
}
```

Now a rep can disagree with something specific. When they say "fit.role is wrong, ops leads at that size absolutely buy tooling", you change one line in a file instead of relitigating the whole model.

Compute the score in code

The agent decides which lines fired. Code adds up the weights. Keep it that way. A model asked to both evaluate criteria and do arithmetic will occasionally produce beautiful reasoning and then hand you a total that doesn't match its own list.

```javascript
const score = rubric.all
  .filter(r => result.fired.includes(r.id))
  .reduce((sum, r) => sum + r.weight, 0);
```

Calibrate against history

Before you trust it on live leads, run the scorer over deals whose outcome you already know. Take last year's closed-won and closed-lost, strip anything the record wouldn't have had at lead time, and score them all.

You want the won deals to cluster high. If they don't, the rubric is wrong, and you've learned that for the cost of a batch job instead of a quarter. I've never had a first rubric survive this test intact, and the fixes it suggests are usually about intent weights being too low.

Run it again after every rubric change. It takes ten minutes and it's the only reason anyone should believe the numbers.

Store the reasoning

Write `fired`, `missed`, and `note` onto the record, not just the score. It costs a text field, and it's the difference between a score people use and a score people route around.

Scored leads get worked. That's where an agent can do real damage, so we go slowly.

Outbound: The Motion That Can Embarrass You

Everything up to here has been internal. Bad enrichment is annoying. A wrong score is a conversation. An email is a thing your customer reads, with your name on it, and you can't take it back.

So the architecture changes here. Drafting and sending become two different systems with two different trust levels.

The drafter never sends

The outbound agent gets `crm-read` and `mail-read` (so it can see the existing thread and avoid repeating what a rep already said) and one write tool: `queue.add_draft`. That's it. It cannot send. Not "is instructed not to send". It doesn't have the tool.

```
drafter    crm-read, mail-read, queue-write
sender     queue-read, mail-send      <- no model in this process at all
```

The sender is a plain script. It reads approved drafts off the queue, checks the preconditions, and sends. There's no reasoning in it, which means there's nothing to talk out of its rules.

Preconditions live in the sender, in code

Every one of these is a hard check in the sending script, never a line in a prompt:

```
async function canSend(draft: Draft, person: Person): Promise<string | null> {
  if (person.unsubscribed) return "unsubscribed";
  if (suppressionList.has(person.email)) return "suppressed";
  if (person.stage === "won" || person.stage === "dead") return "wrong stage";
  if (await hasReplyInThread(draft.threadId)) return "already replied";
  if (await touchesThisWeek(person.id) >= 2) return "touch cap";
  if (!draft.approvedBy) return "not approved";
  if (draft.body.includes("{{")) return "unrendered template token";
  return null;
}
```

That last one is not a joke. The single most common outbound disaster is a template variable that didn't resolve, and it's a three-character check.

Start with a human gate, then narrow it

Week one, every draft gets read by a person before it goes. That's slow, and it's supposed to be. What you're buying is a corpus of "the agent wrote this, the human sent this instead", and that diff is the best signal you'll ever get about your own voice.

After a few hundred, the pattern shows up: some segments the agent gets right basically every time, and some it consistently botches. Auto-approve the first group, keep the gate on the second. That beats a blanket confidence threshold, because the failures cluster by situation rather than by the model's self-assessment.

Rate limits are a safety feature

Cap sends per agent run, per day, and per domain. Not for deliverability, though it helps. A cap is what turns a runaway into a small runaway. An agent with a bad filter that can send 20 emails costs you an apology. The same agent with no cap costs you a domain reputation and a very bad Monday.

Add a kill switch the sender checks on every run: one row in a table, one flag in an env var, something a person who is not you can flip at 2am.

First contact is the risky part. Keeping the conversation alive is mostly a timing problem, and that one automates cleanly.

Follow-Up on a Timer

Follow-up is the motion with the clearest return and the least glamour. Most deals that die of neglect die between touch two and touch three, and the reason is almost never strategy. Somebody got busy.

An agent on a schedule doesn't get busy. But the design work here isn't in the sending. It's in knowing when to stop.

Stop conditions are the feature

Write these down before you write anything else, and encode every one of them as data on the record that code can check:

Stop	Where it comes from	Checked by
Replied	thread has an inbound message	sender precondition
Unsubscribed	link click, header, manual flag	sender precondition
Meeting booked	calendar or CRM stage change	sender precondition
Rep took over	human activity in the last 7 days	sender precondition
Max touches	counter on the sequence record	sender precondition
Explicit no	agent classified the reply as a decline	nurture agent, then flag

Notice that every row lands in the sender's precondition list from the last chapter. The nurture agent proposes. The sender decides whether anything goes at all, because the sender is the thing that can't be argued with.

The loop

The agent runs once a day and asks one question per open sequence: has enough happened, or enough time passed, that the next touch is warranted?

```
// nurture agent, once daily
const open = await crm.listSequences({ status: "active" });

for (const seq of open) {
  const stop = await stopReason(seq);          // code, not model
  if (stop) { await crm.closeSequence(seq.id, stop); continue; }

  const daysSince = daysBetween(seq.lastTouchAt, today());
  if (daysSince < seq.cadence[seq.touchCount]) continue;

  const context = await gatherContext(seq);    // thread, signals, notes
  const draft = await agent.write(seq, context);
  await queue.addDraft(draft);                 // human gate still applies
}
```

The agent's creative job is small: read the thread, read any new signals on the account, and write a touch that reflects what's changed since the last one. That's a genuinely good use of a model. It's the difference between "just bumping this to the top of your inbox" and a follow-up that mentions the three new seats their team added last week.

Classify replies, but hand off the hard ones

When a reply comes in, the agent's job is triage:

```
{
  "intent": "not_now",
  "revisitInDays": 90,
  "confidence": 0.85,
  "note": "Budget frozen until next fiscal year, asked to circle back in Q1."
}
```

Route `interested` and `pricing_question` to a human immediately. Handle `not_now` and `wrong_person` automatically, because those are bookkeeping. Anything under a confidence threshold goes to a human with the reply attached.

I'd let an agent reschedule a lead for ninety days. I wouldn't let it negotiate. The dividing line is whether being wrong costs a calendar entry or a deal.

Outbound and nurture are the motions people build first. The one that pays most reliably is the one nobody wants to build.

Hygiene: The Boring Agent That Pays

If you build exactly one agent from this guide, build this one.

The janitor runs at 6am, reads everything, changes nothing, and files a report. Its blast radius is a Slack message. That makes it the safest possible first agent, and it will find things about your funnel that nobody currently knows.

What it looks for

```
const findings = await Promise.all([
  crm.listStaleDeals({ days: 45 }),
  crm.listDealsWithoutOwner(),
  crm.listDealsPastCloseDate(),
  crm.listPersonsMissing(["title", "accountId"]),
  crm.listLikelyDuplicates(),
  crm.listDeadDomains(),          // MX lookup failed
  crm.listStageRegressions({ days: 30 }),
]);
```

All of that is deterministic and belongs in the server. The agent's job starts after: read the findings, group them into something a human will act on, and write the report.

That framing matters. A raw list of 340 problems gets ignored. A report that opens with "eleven deals worth $240k have had no activity in 45 days, all owned by two reps who are both on the enterprise team" gets a response before lunch.

It files, it doesn't fix

The janitor proposes changes into a queue, with a diff:

```
{
  "kind": "merge_duplicate",
  "primary": "per_8812",
  "duplicate": "per_9403",
  "reason": "Same person, personal email on the duplicate, no activity since creation",
  "diff": { "email_alt": "+ sam@example.com", "activity": "merge 2 events" },
  "confidence": 0.92
}
```

Approve in bulk, reject the ones that look wrong, and let a dumb script with `crm-write` apply them. Same shape as outbound: reasoning in one process, mutation in another.

After a month of approvals, look at what you've been rubber-stamping. Whole categories will be at 100% (dead domains, missing owner on closed-lost), and those can move to automatic. The categories where you keep overriding stay manual, probably forever, and that's fine.

The report is the product

Send it somewhere people read. A daily message with a short summary, the top five things worth a human's attention, and a link to the full queue. Keep the format identical every day so it becomes scannable.

One thing I'd add from experience: track the numbers over time and put the trend in the report. "Stale deals: 41 (was 63 last week)" tells a manager something. "Stale deals: 41" tells them nothing.

Why this one pays

Hygiene work is real revenue. Deals rot because nobody noticed. Duplicates split activity history, so your scorer reads half the signal. A missing owner means an inbound lead sat for six days. None of it is interesting, all of it costs money, and it's precisely the work humans deprioritize and machines don't.

The janitor watches your CRM. The next agent watches your product, which is where the honest signals live.

Signals: Expansion and Churn

Your CRM records what your team did. Your product records what your customer did. The second one is more honest, and most GTM automation ignores it entirely.

Wire your product analytics in as a read-only MCP server and a whole class of agent becomes possible: one that notices things.

Define the signal in code

This is the rule I'd put on the wall: never let a model compute the number it's supposed to be alarmed about.

```
// analytics-read server
async function weeklySignals(accountId: string): Promise<Signal[]> {
  const now = await usage(accountId, lastNDays(7));
  const prev = await usage(accountId, priorNDays(7, 7));
  const out: Signal[] = [];

  const drop = (prev.events - now.events) / Math.max(prev.events, 1);
  if (drop > 0.4)
    out.push(sig("usage_drop", drop, `${pct(drop)} fewer events week over week`));

  const seats = await seatUsage(accountId);
  if (seats.active / seats.licensed > 0.9)
    out.push(sig("seat_limit", seats.active / seats.licensed,
      `${seats.active}/${seats.licensed} seats active`));

  const champ = await lastSeen(accountId, "champion");
  if (daysSince(champ) > 21)
    out.push(sig("champion_inactive", daysSince(champ), `Champion last active ${champ}`));

  return out;
}
```

Arithmetic in code, every time. The threshold is a business decision that belongs in a config file where someone can tune it. The calculation is a calculation.

What the agent adds

Given a stream of signals, the agent does three things a query can't.

It writes the narrative. Three signals on one account is a story. "Usage down 47%, champion inactive 24 days, renewal in 60 days" is a paragraph a human acts on. Three rows in a table are three rows in a table.

It separates the alarming from the seasonal. A 40% usage drop the week of Christmas is not a churn signal. An agent with the account's history and a calendar can say so, and can flag when the same account's drop in March is genuinely different.

It routes. Expansion signals go to the account owner with a suggested next step. Churn signals go to CS with a suggested next step. Signals on accounts under $5k a year go into a weekly digest, because attention is finite and that's a business rule the agent can apply.

Expansion is the easier win

Everyone builds churn detection first. Expansion detection is less work and pays faster, because those accounts are already happy and the signals are unambiguous. Seats near the limit, usage past a plan tier, a new team in a new department showing up in the logs. Every one of those deserves a conversation, and nobody is having it because nobody knows.

```
signal: seat_limit, 0.94, "47/50 seats active, up from 31 last month"
route:  account owner
draft:  "Noticed you're at 47 of 50 seats..."   <- drafter agent, human gate
```

Same pipeline as outbound. Same guardrails. Different trigger.

Feed signals back into scoring

Signals belong in the rubric too. A prospect account with a pricing page visit and two signups this month should outrank an identical account with neither, and that only happens if your scorer reads the signal stream from Chapter 3.

Signals tell you what's happening now. The next question every executive asks is what happens next quarter, and that one needs care.

Forecasting You Can Defend

Ask a language model to forecast your quarter and it will give you a number. The number will be confident, well-formatted, and made up. Forecasting is the place where the wrong architecture produces output that looks the most like the right answer, which makes it the most dangerous motion in this guide.

The fix is the same as everywhere else, applied harder. Numbers come from code. The agent writes the memo.

Snapshot in code

```
// runs before the agent, deterministic, no model involved
const snapshot = {
  quarter: currentQuarter(),
  committed: sum(deals.filter(d => d.forecastCategory === "commit")),
  bestCase: sum(deals.filter(d => d.forecastCategory === "best_case")),
  weighted: deals.reduce((s, d) => s + d.amount * probability(d.stage), 0),
  closedToDate: sum(deals.filter(d => d.stage === "won" && inQuarter(d.closedAt))),
  target: TARGETS[currentQuarter()],
  byStage: groupSum(deals, "stage"),
  slipped: deals.filter(d => d.closeDate < today() && d.stage !== "won"),
  agedOver60: deals.filter(d => daysInStage(d) > 60),
};
```

Hand that object to the agent whole. Tell it, explicitly, that it may not compute new totals and may not state a number that isn't in the snapshot. Then give it read-only tools so it can go look at individual deals: activity history, last touch, notes, signals from the last chapter.

Ask for judgment, not math

The prompt is the interesting part:

> Here is this quarter's pipeline snapshot, plus read access to the deals in it. Do not compute any totals; use only the figures given.
>
> 1. Name the five deals most likely to slip, with the specific evidence from their activity, notes, or signals.
> 2. Name any deal marked commit that you'd question, and say why.
> 3. Point out anything in the shape of the pipeline that a manager should look at.
> 4. Keep it under 400 words.

What comes back is genuinely useful, because "this deal is marked commit, the last customer-side activity was 31 days ago, and the champion's usage signal fired last week" is a pattern-match across four systems that no human does routinely and no query can phrase.

The deliverable is a memo

Not a number. Not a dashboard with an AI badge on it. A short written argument, dated, saved, with the snapshot attached.

Save both. Six weeks later you can go back and check whether the deals it flagged actually slipped. That's the only way to find out if this agent is worth its tokens, and it's a check almost nobody runs.

I've had a forecaster call three of five slips correctly in a quarter and be flatly wrong about the other two, in

a way that told me the rubric was overweighting recency. Both halves of that were useful. Neither would have been visible if I'd kept only the number.

What it must never do

No writes. The forecaster gets `crm-read` and `report-write` and nothing else. A forecasting agent that can update `forecastCategory` eventually will, and then your forecast is a forecast of itself.

Every agent in this guide now exists. Before they all run at once, we need to talk about what happens when one of them is wrong.

Guardrails and Blast Radius

Every agent in this guide will be wrong sometimes. Design around that instead of hoping otherwise.

The question for each motion is simple: when this is wrong, what does it cost, and who finds out?

Motion	Wrong looks like	Blast radius	Gate
Janitor report	Bad summary	Someone reads a dull message	None
Enrichment	Wrong job title	A rep is mildly misled	Provenance, rollback
Scoring	Good lead scored low	A deal is worked late	Reasoning stored, rep can override
Capture merge	Two people merged into one	Lost history, hard to unwind	Confidence threshold, queue
Nurture timing	Touch a day early	Nothing	None
Stage writes	Deal moved wrongly	Forecast is off	Queue, audit log
Outbound send	Bad email to a real customer	Reputation, trust, the deal	Human approval, hard preconditions, caps

Read down that last column. The gating is deliberately uneven. Gate the janitor's report the way you gate a send and nobody will use either one.

The five rules

Capabilities beat instructions. If an agent must not send email, don't tell it not to send email. Don't give it the tool. Anything a prompt forbids is negotiable by anything that reaches the context window, including a lead's job title field.

Dry run is the default. Every write-capable agent takes an `--apply` flag, and it defaults off. New agent, new prompt, new model version: run it dry for a week and read the diffs. This has caught more bugs for me than every test I've written for these systems.

Everything reversible, everything logged. One append-only table:

```
type AgentAction = {
  agent: string;
  runId: string;
  at: string;
  target: string;      // "person:8812"
  before: unknown;
  after: unknown;
  approvedBy: string | null;
};
```

With `before` stored, "undo everything the enricher did on Tuesday" is a script. Without it, that sentence is a project.

Caps on everything that leaves the building. Sends per run, per day, per domain. API spend per agent per day. Records modified per run. A cap turns a runaway into an incident report instead of an outage.

A kill switch a non-engineer can reach. One flag, checked by every agent at the top of every run, flippable from a UI or a chat command. If turning off your agents requires a deploy, they aren't production systems.

Separate credentials, always

One token per agent, scoped to exactly its tools. It's more setup, and it buys two things. Your audit log shows which agent did what without you having to trust the agent to report it, and revoking a misbehaving agent takes seconds and affects nothing else.

Guardrails in place, we can finally let the whole thing run.

The Daily Run

Here's the whole system in one day. This is the schedule I'd start with, and the ordering matters more than the exact times.

```
:00 every 15 min    capture      new leads in, deduped, normalized
06:00               janitor      overnight hygiene scan, report + queue
06:30               enricher     fill records that will be scored today
07:00               scorer       rescore anything enriched or changed
07:30               signals      product signals, routed
08:00               drafter      outbound + nurture drafts into review queue
09:00               sender       approved drafts go out
hourly 09-17        sender       approvals from during the day go out
18:00               digest       what happened today, what's waiting
Mon 08:00           forecaster   weekly pipeline memo
```

Why this order

Each agent depends on the one before it. Enrichment before scoring, because a scorer reading an empty record produces confident nonsense. Scoring before drafting, because the drafter picks who to write to off the score. Signals before drafting, so a touch can mention the thing that just happened. Sending after drafting with a gap, because the gap is where a human reads the queue.

Hygiene goes first, at 6am, so it catches yesterday's mess before today's agents build on top of it.

Make runs idempotent

Every agent must survive being run twice. Someone will trigger a manual run. A cron will double-fire. A retry will land after a partial success.

```
const runKey = `${agent}:${target}:${dayStamp()}`;
if (await runs.has(runKey)) return skip("already ran today");
```

For the sender, idempotency is a hard safety requirement. Guard sends on a message-level key rather than a run-level one, and write the key before you send rather than after. A duplicate skip is invisible. A duplicate send is an apology.

Failures should be loud and local

An agent that throws should fail its own run, log it, and not block the next agent in the chain. The scorer dying at 7am is no reason for the janitor's report to go missing. Wrap each run, catch, log, alert, continue.

What you want out the other end is a daily digest a person actually reads:

```
GTM agents, Tuesday
  captured    38 leads (4 merged as duplicates, 2 queued for review)
  enriched    22 of 34 eligible ($4.10 provider spend)
  scored      41 (7 above threshold, routed to owners)
  signals     3 expansion, 1 churn risk (Acme, usage -47%)
  drafts      9 queued, 6 approved and sent, 3 waiting on you
  hygiene     11 proposals queued, 41 stale deals (was 63 last week)
  errors      enricher: provider B timeout on 2 records, retried
```

That message is the interface to the whole system. If you can read it in fifteen seconds and know whether anything needs you, the architecture is working.

Now the last question: how do you get here without breaking the thing that pays everyone's salary?

Rolling It Out Without Breaking Revenue

The failure mode I've watched most often is a good agent shipped into a live funnel on a Tuesday with no way to tell whether it helped.

Go in this order. It's sorted by blast radius, smallest first, which also sorts it from "nobody will notice if it's wrong" to "everybody will notice if it's wrong".

The order

1. **Janitor, report only.** Two weeks. Change nothing. Read the reports and find out what's actually wrong with your data.
2. **Janitor with a queue.** Now it proposes fixes and you approve them. You'll learn which categories you always approve.
3. **Enrichment, dry run.** Log what it would write for a week. Diff it against reality. Then turn on writes with provenance.
4. **Scoring, shadow mode.** Score every lead, store the score, show it to nobody. Compare against what reps actually worked and what actually closed.
5. **Signals.** Read-only by nature. Route to humans, let them act.
6. **Nurture, human gate.** Every touch approved before it goes.
7. **Outbound, human gate.** Same, on the higher-stakes motion.
8. **Selective auto-approve.** Only for segments where your approval rate has been essentially 100% for a month.

Most teams should stop somewhere around step six and be very happy. The last two are optimizations, and they're the ones with teeth.

Shadow mode is the whole trick

For any agent that scores, ranks, or decides, run it in parallel with the humans for a few weeks and compare. It costs almost nothing, it's the only real evidence you'll get, and it turns "the AI thing seems good" into a number you can show a skeptical VP.

Scoring is the clearest case. Score every lead, hide the number, and after six weeks ask: of the leads that closed, where did they rank? If your winners are scattered evenly across the range, your rubric is decorative, and now you know before you've routed a single deal by it.

Measure the funnel, not the agent

Token spend and run counts are operational metrics. They tell you nothing about whether this worked. Watch these instead:

- Time from lead creation to first human touch
- Percentage of inbound that gets touched at all
- Duplicate rate in the CRM
- Deals aging past 45 days with no activity
- Reply rate on agent-drafted versus human-drafted email
- Hours per week the team spends on CRM admin

Baseline every one of them before you ship anything. You can't prove an improvement you didn't measure first, and "it feels faster" loses every budget conversation.

When to turn one off

Kill an agent when its approval rate drops below about 70% and stays there, when its queue goes unread (which means nobody trusts it), or when the thing it automates changes shape. Turning one off is cheap, and being willing to do it is what makes shipping the next one easy.

The team you want at the end of this is one where nobody spends Friday afternoon deduping records, and pipeline reviews start from a memo somebody already read.

Build the janitor this week. Everything else follows from having one agent you trust.

Toolkit: The GTM Agent Checklist

Print this. Work down it per motion.

Before the first agent

- Every system you'll touch is wrapped as an MCP server, named for the funnel rather than the vendor
- Read tools and write tools are separate servers with separate credentials
- A canonical `Person` / `Account` / `Signal` shape is defined and written down
- Every field an agent writes carries `source`, `confidence`, `updatedAt`
- Human-entered values are protected from agent overwrite, in code
- The CRM field mapping lives in exactly one file
- An append-only action log with `before` and `after` exists
- A kill switch exists, and someone who isn't you can flip it

Per agent

- It does one motion
- Its tool allowlist is the minimum it needs, and nothing it must not do appears in it
- It has its own credential
- It has an `--apply` flag that defaults to off
- It's idempotent, keyed on something stable
- It has a cap: records touched, spend, or sends
- It fails loudly and doesn't block the next agent
- Its output includes reasoning, not just a result
- Someone read a week of its dry-run output before it went live

Anything that touches a customer

- The drafting process and the sending process are different processes
- The sending process contains no model
- Unsubscribe, suppression, stage, reply, touch cap, approval, and unresolved-template checks all run in the sender, in code
- Sends are capped per run, per day, and per domain
- The first few hundred went through a human, and someone read the diffs

Ongoing

- Baseline funnel metrics were captured before anything shipped
- The rubric is recalibrated against closed-won after every change
- The forecaster's memos are saved and checked against outcomes
- Approval rates are tracked per agent and per category
- The daily digest is short enough that someone actually reads it

The four rules, if you remember nothing else

1. Arithmetic in code. Judgment in the model. Never the reverse.
2. Capabilities, not instructions. If it must not, it can't.
3. Provenance on every write, so every write is reversible.
4. The process that reasons is never the process that sends.

Start with the janitor. Ship it Monday, read its reports for two weeks, and let it tell you which of the other agents your funnel actually needs.

If you want the protocol layer underneath all of this, MCP from Scratch (/guides/mcp-from-scratch) builds a server from an empty folder. If you want to run these seven as a coordinated fleet instead of seven crons, Running the Fleet (/guides/running-the-fleet) covers the orchestration.

About Roger

I'm Roger Stringer. I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I work as a fractional CTO through **Data McFly** (https://datamcfly.com), helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. Pointing agents at a funnel is a good chunk of that work, and it almost always opens on the two questions this guide answers the long way: clean your data before you automate it (https://datamcfly.com/blog/clean-your-data-before-you-automate), and pick the one workflow to start with (https://datamcfly.com/blog/start-with-one-workflow-how-to-pick-your-first-automation). If you'd rather not wire the whole pipeline together yourself, book a free 30-minute call (https://datamcfly.com/how-it-works). No pitch, no pressure.

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email**: roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X**: @freekrai (https://x.com/freekrai)
- **GitHub**: github.com/freekrai (https://github.com/freekrai)
- **LinkedIn**: linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting. Follow along wherever suits you.