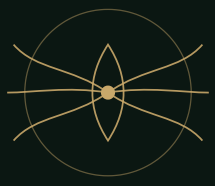

THE BRAKE

How to Run Autonomous Agents Without Going Broke

The cost everyone watches isn't the one that gets you. Here's the guardrail I put under every agent before I let it run on its own — so it can work all night and never spend a dollar I didn't allow.

BY **HUNTER GRANT** · HUNTERGRANT.DEV

BUILT WITH CLAUDE — FABLE 5
DAY 9 OF BUILDING IN PUBLIC

START HERE

The bill that kills you isn't the one you're *watching*.

Everyone building with AI watches the token meter. It's the wrong meter. Across a real run of my swarm, the language-model tokens came to pennies — a few dollars for a night of work. Tokens are not the threat.

The threat is the moment an agent does something in the real world on a paid rail — places a phone call, hits a metered API, sends mail at scale. Those cost 15–30× what the thinking did. An agent that can take those actions on its own is a liability the day it's wrong — unless you put a brake under it first.

> *Tokens are pennies. It's the real-world actions that bankrupt you.*

Four parts, all deterministic — no model gets a vote on whether to spend. Generalized so you can bolt them onto any agent before you walk away from it.

THE FOUR — AND ONE MORE

01	The Real Unit	FIND WHAT ACTUALLY COSTS MONEY
02	The Ledger	MEASURE EVERY CENT BEFORE YOU CAP IT
03	The Brake	A GATE WITH NO MODEL IN THE DECISION
04	The Switches	COSTLY RAILS OFF BY DEFAULT
+	The Cheap Seat	THE RIGHT MODEL FOR THE JOB

01 The Real Unit

Find what actually costs money. It's not tokens.

USE IT WHEN

Before you cap anything. You can't protect a budget until you know which action on it is the expensive one — and it's almost never the model.

THE NUMBERS

```
-- a night of real agent work, by cost line:

LLM TOKENS        ~$1-3 per batch     // the part you watch
                  pennies per clean run

A PHONE MINUTE    ~$0.12-0.31 / min   // 15-30x the tokens

A METERED SCRAPE  per-record / per-call

A MASS SEND       deliverability + spend at volume

-- the rule: your COGS is the cheapest real-world action
-- your agent can take, times how often it can take it.
-- meter THAT as a first-class number. tokens are noise.
```

> **Why it works.** Optimizing token cost is rearranging pennies while a dollar walks out the door. The expensive thing is the agent *acting* — a call connected, an API charged, a thousand emails sent. Name that unit out loud and the whole design reorients around protecting it, instead of around a meter that was never going to hurt you.

02 The Ledger

You can't cap what you don't measure.

USE IT WHEN

The instant an agent can spend. Every real-world action writes one immutable row before it's allowed to repeat — an append-only record of money out the door.

SCHEMA · APPEND-ONLY

```
-- an irrevocable record of every costly action.
-- append-only: no UPDATE, no DELETE. it can only grow.

create table spend_ledger (
    id          uuid primary key default gen_random_uuid(),
    source      text not null, -- call | scrape | send | tokens
    units       numeric not null, -- minutes, records, messages...
    cost_usd    numeric not null default 0,
    occurred_at timestamptz default now()
);

-- the two numbers the brake reads back, fast:
-- today's $ spent  (sum where occurred_at >= day start)
-- this month's COGS (sum where occurred_at >= month start)
```

Why it works. Append-only is the whole trick: a ledger you can edit is a ledger you'll quietly fudge at 2 a.m. when the agent's on a roll. Make the record immutable and the spend becomes a *fact*, not an opinion. Now the brake has real numbers to read — today's spend and this month's — instead of a guess.

03 The Brake

A gate with no model in the decision.

USE IT WHEN

Before every costly action. The agent doesn't get to decide if it can afford something. Plain code checks the caps against the ledger and returns yes or no.

CODE · DETERMINISTIC

```
// caps live in a config table only YOU can write.
// conservative defaults: phone $ cap = 0 (blocked).
caps = { perRunMaxUsd: 0.50, dailyTokenUsd: 5,
         monthlyMinutes: 0, monthlyPhoneUsd: 0 }

// the gate runs BEFORE the action. no LLM. pure math.
function phoneAllowed(state, addMin, addUsd) {
   if (caps.monthlyMinutes <= 0)  return no("disabled")
   if (state.monthMin + addMin > caps.monthlyMinutes)
                                  return no("minutes cap")
   if (state.monthUsd + addUsd > caps.monthlyPhoneUsd)
                                  return no("$ budget")
   return yes()
}

// over budget? the action never happens. it doesn't ask twice.
```

> **Why it works.** The decision to spend can never be left to the thing doing the spending. A deterministic gate — config caps in, ledger totals in, yes/no out — can't be talked into an exception, can't hallucinate headroom, and reads the same under load as it does in a demo. The model proposes the action; *code* decides if it's affordable.

04 The Switches

Costly rails off by default. On only on purpose.

USE IT WHEN

From the first deploy. The agent ships fully wired but with every real-world rail switched OFF — it can think, score, and draft, but it cannot send until you flip a flag.

PATTERN · SHADOW MODE

```
-- one row per rail. all default FALSE. operator-only writes.
insert into flags (key, enabled) values
   ('engine_live', false), -- master: loop runs at all
   ('email_live',  false), -- may actually send mail
   ('phone_live',  false); -- the costliest rail. stays off.

-- in code, before any send:
if (!flags.phone_live) { bankProposal(draft); return; }
-- shadow mode: it does ALL the work, then files a proposal
-- instead of firing. you flip a flag the day you trust it.
```

> **Why it works.** "Off by default" means a bug, a bad prompt, or a 3 a.m. edge case can't cost you anything — the worst case is a draft you don't send. You get to watch the agent behave for a week with the rails cold, see exactly what it would have done, and arm one rail at a time. Trust is earned in *shadow mode*, not assumed in production.

BONUS · THE LEVER ONCE THE BRAKE HOLDS

+ The Cheap Seat

The right model for the job — not the best one for all of them.

USE IT WHEN

Once spending is safe, make it efficient. Most agent work is grunt work. Route it to a cheap model; save the expensive, top-tier model for the few calls that actually need it.

PATTERN · TIERED ROUTING

```
-- one router, four tiers. pick by the job, not by habit.
route(job):
   GRUNT   parse, classify, dedupe   cheapest model
   REASON  score, draft, summarize   mid model
   APEX    anything customer-facing  top model
           or touching private data

-- the discipline that matters more than the price list:
-- cheap models for non-sensitive grunt work;
-- the trusted model for anything with real names,
-- real money, or a real person on the other end.
```

> **Why it works.** Paying top-tier rates to deduplicate a list is the same mistake as the token panic, in reverse — spending where it doesn't buy you anything. Tiering cuts the bill several-fold with no quality loss on the work that's visible, because the visible work still rides the best model. Cost discipline isn't spending less; it's *spending where it counts*.

THE WHOLE BRAKE

Meter the real unit. Bank it. Gate it in code. *Default it off.*

Do those four and an agent that runs all night stops being a risk and starts being an employee — one that physically cannot spend a dollar you didn't allow. The brake isn't what slows you down. It's what lets you leave the room.

— *Hunter*

FOLLOW THE BUILD

@huntergrant.dev

Building a system you want a second set of eyes on? That's the work. The door's open.