A FIELD GUIDE • FOR THE AI-NATIVE BUSINESS

Give your agent a *nervous system*

Your agent already has a brain. This is the story of the system that runs underneath it — the one that lets it work while you sleep.

15 concepts · 3 layers · 1 idea

From an agent you operate → a worker that runs itself

THE GAP

You built an agent that works. The trouble is it only works *while you watch it.*

You open your agent, you type, it replies. Step away and it stops. Closing that gap — between an agent you *drive* and a worker that *runs on its own* — is the whole job.

What closes the gap is not a smarter agent. It is a missing system.

A PICTURE YOU ALREADY OWN

THE SYSTEM YOU NOTICE

Your brain thinks. Your muscles act.

You decide to reach for the cup, and you reach. It is deliberate, and it needs your attention.

THE SYSTEM YOU DON'T

A second system runs underneath.

Your heartbeat, your reflexes, the signals that keep you alive while you sleep. Stop paying attention — your heart keeps beating.

That second system is the autonomic nervous system. It is the difference between a body and a corpse.

THE DIAGNOSIS

Your agent has a brain. It has *no nervous system.*

It has everything it needs to *think* : a model to reason, tools and skills to act. What it lacks is the system that runs when you look away.

So the moment you stop driving it, it stops. No heartbeat. No reflex. No way to sense the world or hold its place while it waits.

This course gives the agent that missing system — without changing the agent.

01

It senses

Wakes the agent when something happens in the world.

02

It reflexes

When a step breaks, it recovers without redoing the work that already finished.

03

It rests

Waits hours for a person or a slow API at zero compute.

04

It balances

Keeps steady when five hundred requests arrive at once.

It closes the loop on its own. That is the line between *an agent you operate* and *an FTE that runs itself.*

THE ONE IDEA THIS RESTS ON

You *add* a nervous system. You do not rewrite the agent.

PROGRAM ONE — YOU DON'T WRITE IT

The engine

A *durable execution engine* . It catches events, retries, remembers every step, waits, and shows you a dashboard.

→

PROGRAM TWO — THE ONLY PART YOU WRITE

Your agent

It thinks and acts — exactly as it does today. The engine reaches it over a thin web wire and drives it for you.

We use Inngest as the concrete engine. The same patterns run on Temporal, Restate and Dapr. Day AI, a CRM for AI-native companies, calls Inngest “the nervous system” of their product.

THE MAP • 15 CONCEPTS, THREE JOBS

Three layers do the work

01 — SENSES

Triggers

How the world reaches the worker — events, schedules, webhooks, fan-out.

02 — REFLEXES

Durable execution

Keeping it correct when a step breaks — checkpoints, memory, retries.

03 — BALANCE

Flow control

Staying healthy under load — limits, fairness, recovery, the human gate.

Senses wake it · reflexes carry it through failure · balance keeps it steady · and a gate lets a human decide.

LAYER ONE

I The Senses

How the world reaches the worker

An agent you call by hand runs when you call it. A real worker has senses — it runs when the *world* reaches it. A customer emails. A clock strikes nine. A payment fails. Another worker hands off.

Events, not requests

A *request* is a conversation someone waits on. An *event* is a note the world leaves behind — and moves on.

Once you think in events, durability and scale fall out almost for free: the producer never waits, a crashed worker retries off the stored event, and new workers can listen without anyone rewiring.

IN THE BUSINESS

01

A customer emails support. With a request , someone—or some server—sits on hold until the reply is drafted.

With an event, the inbox just records “ email received ” and lets go. The worker picks it up on its own time.

`customer/email.received → fire & forget`

WHY THE SHIFT MATTERS

• 50 EMAILS A MINUTE, 8 SECONDS EACH

REQUEST MODEL

~7

handlers held open just to keep up

VS

EVENT MODEL

1

parser. It fires and returns in ten milliseconds.

The event queue absorbs the spike. Events are how you stop owning the timing of work.

Work that runs because *time passed*

The simplest sense is the clock. Not a reaction to the world — just the calendar coming around.

```
# every day at 9am cron("0 9 * * *")
```

IN THE BUSINESS

02

Every morning at nine, a customer-health report lands in the team's Slack — nobody clicked a button.
Weekly cleanups. Hourly recalculations. The standing chores of a business, on a timer.

scheduled · no trigger but the time of day

When the outside world calls in

A webhook is the doorbell. Some other system — Stripe, your email host, a form — pushes a message *to* you.

The inbound payload becomes a named event, and your worker reacts to the name. You wire it once in a dashboard, not in code.

IN THE BUSINESS

03

A customer's card fails to charge. Stripe rings the doorbell the instant it happens.

Your worker wakes, looks up the account, and starts the recovery flow — before anyone notices revenue slipping.

```
stripe: charge.failed → named event
```

When the same event arrives *twice*

The same message will sometimes reach you twice. A unique ID makes the duplicate a quiet no-op.

Idempotency is a word for one promise: do this once, no matter how many times the message shows up. It is what keeps retries safe.

IN THE BUSINESS

04

A customer clicks “Issue refund.” The page hangs. They click again.

Two events arrive — but the card is refunded exactly once. The second click lands on a closed door.

```
same id → second delivery dropped
```

One signal, many workers

A single event can wake a whole department at once — each part reacting on its own, none waiting on the others.

One event, N listeners; or one parent firing N child events. This is how a scheduled wake-up becomes hundreds of independent runs.

IN THE BUSINESS

05

A refund fails. That one fact needs to reach four places at once:

- notify the support agent
- write the audit trail
- update the customer’s risk score
- alert finance

`one event → four reactions, in parallel`

LAYER TWO

II The Reflexes

What happens when something breaks

The senses got work to the worker. The reflexes are what happen when that work breaks halfway through — and the worker carries on anyway, without losing what it already did.

Every step is a *checkpoint*

A normal program that crashes starts over from the top. A durable one resumes from the last step that finished.

```
step.run("charge-card", …) # saved the moment it succeeds
```

IN THE BUSINESS

06

A six-step order flow crashes at step four. The naive version re-runs steps one through three — charging the card again. The durable version picks up at step four , the first three already banked. The customer never feels the crash.

crash between steps → resume, don’t restart

WHY THIS IS NOT OPTIONAL

• A SIX-STEP WORKFLOW AT 95% PER STEP

WITHOUT REFLEXES

26%

chance the workflow fails somewhere

→

WITH MEMORY + TARGETED RETRIES

99.7%

overall reliability, same agent

Small per-step failures compound . Reflexes are what stop a chain of likely steps from becoming an unlikely whole.

Done once stays done

The mechanic underneath it all: a completed step hands back its stored answer instead of running again.

This is *memoization*. When a later step fails and the run retries, every finished step is already paid for — it just returns its result.

IN THE BUSINESS

07

Two costs you never want to pay twice: a card charged again, and an expensive model call re-run on every retry.

Memoization pays each exactly once. The retry only re-does the one step that actually broke.

```
finished step → returns from memory, free
```

Durability through *time*

Some work has to wait — for three days, or for a person. The worker can suspend, durably, and pick up where it stopped.

```
step.sleep("3 days") · step.wait_for_event(…)
```

IN THE BUSINESS

08

A new customer signs up. Send the welcome email now — then wait three days and send the follow-up.
One function holds the whole story across three days . No cron table, no reminder queue, no glue.

suspend for a duration — or for an event

THE PART THAT SURPRISES PEOPLE

A worker can wait *three hours* for a human and burn *zero compute* the entire time.

It is not a process sitting in a loop, holding a connection, costing money. The engine simply holds the resume time and wakes the worker when the moment comes. Waiting is finally free.

The reflex in close-up

A step fails, the engine waits a moment and tries again — longer each time. After a few tries, the failure is parked, not lost.

Automatic backoff retries by default. Whatever can’t recover lands in a dead-letter state, kept whole for you to replay later.

IN THE BUSINESS

09

A payment API blips for ninety seconds. Without reflexes, every order in that window is just lost.
With them, each order quietly retries and succeeds on attempt three. Nobody writes a support ticket.

```
~4 retries, backoff → then dead-letter
```

The thinking, made *durable*

Wrap the model call in the same checkpoint as everything else. Nothing special — the agent's reasoning becomes crash-safe for free.

```
step.run("draft-reply", agent.run)
```


10

The agent spends real money and real seconds drafting a reply. That is the expensive step.
Inside a checkpoint, that draft is written once and remembered. A later failure never makes the agent think the same thought twice.

---

the agent never imports the engine

LAYER THREE

III Balance & Recovery

Staying healthy under real load

A worker that runs and survives crashes still has to behave under load — not melt the systems around it, not let one loud customer starve the rest, and recover fast when a whole batch goes wrong. This is what makes it safe to put in front of paying customers.

Cap the rush

Two knobs: *concurrency* caps how many runs go at once; *throttle* caps how many *start* per minute.

None of this lives in your code — it is configuration the runtime enforces. Writing the same fairness by hand is a queue, a scheduler and a rate limiter, hundreds of lines.

IN THE BUSINESS

11

A busy Monday drops 1,000 emails at 9am. The worker tries to open 1,000 model calls and 1,000 database connections at once. A cap turns the flood into an orderly line. The work all gets done — nothing tips over.

match the throttle to your real API limit

A fair share for everyone

Priority orders the queue. A per-customer cap means no single account can hog the whole worker.

This is multi-tenant fairness: the thing that keeps a shared worker from feeling broken to everyone whenever one tenant gets busy.

12

One enterprise customer fires off five hundred requests. Should every hobbyist now wait behind them?
No. The noisy account takes a slot or two — and everyone else keeps moving.

per-key concurrency → no one starves

Bulk work, done *cheaply*

Some work is naturally grouped. Accumulate many events into one batched call instead of paying per item.

Fewer, bigger operations: fewer model calls, fewer writes, lower cost — without changing what the work actually is.

13

You need to summarize 10,000 conversations. One model call each is slow and expensive.
Group them fifty at a time. Same result, a fraction of the calls.

`accumulate → one batched call`

An undo button for production

Replay failed runs on new code. Bulk-cancel the ones you no longer want. Recovery becomes a few clicks, not a long night.

Because every step was recorded, the engine knows exactly what failed and where — so it can re-run precisely the work that needs it.

You ship a bug. A thousand runs fail in six hours before anyone notices.

You fix the code and replay the thousand.

The lost afternoon repairs itself.

`fix → replay failed runs on new code`

BALANCE • CONCEPT 15 · THE KEYSTONE

The agent *proposes.* A human *approves.*

IN THE BUSINESS

A customer is owed a $500 refund. The agent investigates and decides it's warranted — but it must not issue it alone.

It pauses. A person clicks approve. The agent wakes and finishes. Or no one answers in four hours — and nothing is paid.

WHAT IT REPLACES

Without this, you build the approval queue yourself: a database table, a polling loop, timeout handling, an audit trail. *That is a project, not a feature.*

```
step.wait_for_event
```

This is where the human mind re-enters the loop on a high-stakes action — durably, for as long as it takes.

THE WORKED EXAMPLE

IV From an agent to a worker

Seven plain-English prompts to your companion agent

You don’t write this by hand. You direct your general agent — your companion — in short prompts, and it writes the code. Everything so far was the model. Here you assemble the real thing, one layer at a time.

THE BRIEF • SMALL ON PURPOSE

A customer-support worker

01 Reads its sample customers from a database — id, email, tier.

02 Drafts a warm reply to an incoming customer email.

03 Issues a refund only with human approval.

04 Writes an audit row for every action it takes.

The agent is the easy part — so we keep it small, and spend all our effort on the nervous system around it.

ONE EMAIL, END TO END

The engine drives the worker, one step at a time

1. A customer emails → the engine catches the event
2. Audit: “message received” · load the customer
3. The agent drafts a reply – the thinking part, made durable
4. Is it a refund? Pause for a human – waits hours, survives crashes
5. On approve, issue the refund · on reject, record it
6. Audit: “reply sent”

If a step crashes, the engine re-runs only that step — never the finished ones.

THE SHAPE OF THE BUILD • THE AGENT NEVER CHANGES AFTER D0

Seven prompts, one layer each

D0	D1	D2	D3	D4	D5	D6
The worker built once, standalone	Reflex make it durable	Sense wake on an email	Sense a daily cron, fanned out	Balance flow control	Gate the keystone	Proof survive a broken step

D0 is the agent. Every layer after it is the nervous system — added from the outside.

First it exists, then it survives, then it listens

D0 · THE WORKER

“Build me a minimal support agent. It reads customers, drafts a reply, and the refund tool needs approval before it runs.”

Plain Python. It never imports the engine.

D1 · MAKE IT DURABLE

“Wrap the agent call in a durable function so it survives crashes and the whole call is memoized.”

The expensive thinking is now paid for once.

D2 · WAKE ON AN EVENT

“Make the worker wake on an email event, with an audit row before the agent and after it.”

The opening picture, now running for real.

Each prompt is one sentence. The companion agent handles the detail.

SCHEDULED WORK & LOAD • D3 — D4

Now it runs on a schedule — and under pressure

D3 · A DAILY CRON THAT FANS OUT

A 9am cron fires one health-check per Pro and Enterprise customer — each its own durable run, each idempotency-keyed so a re-delivered cron never double-fires.

`one wake-up → N independent runs`

D4 · FLOW CONTROL

A global cap, a per-customer cap, and a throttle. Fire twenty events across five customers and they queue under the cap — none dropped, none duplicated.

`three decorator arguments, not three hundred lines`

Same agent inside each path. Only how the world reaches it differs.

THE KEYSTONE • D5 · A DURABLE APPROVAL GATE

The pause that survives a crash, a deploy, and a reviewer at lunch

- The agent decides a refund is warranted — and pauses.
- The function sleeps — free while it waits, for minutes or hours.
- A human clicks approve or reject — the decision event arrives.
- The function wakes and finishes — the refund fires at most once, or never.

No queue, no polling loop, no “is it approved yet?” flags. The runtime holds the pause for you.

Break a step, and watch the memory hold

Deliberately break the agent step. Fire an email. In the failing run, the audit step shows *one* completed attempt — its row written once — while the agent step climbs through several retries before it fails.

That gap — one step holding at a single attempt while the broken one accrues many — *is* memoization, in your own worker.

INSIDE ONE FAILING RUN

audit step	**1 attempt**
agent step	**4 attempts → fail**

Then fix the step and replay — the work repairs itself on a fresh run.

A worker that loses work on a bad deploy is just an agent you call. One that fails loudly, retries cleanly, and recovers – is an AI worker.

WHAT JUST HAPPENED

The agent code never changed. *Its reach did.*

You started with an agent you operate — prompt it, watch it, prompt it again. You now have a worker that operates on its own: the world wakes it, its reflexes carry it through failure, it holds its balance under load, and a human steps in only where the stakes demand one.

THE ECONOMICS • TWO COST SURFACES

What this costs to run

INFRASTRUCTURE

Roughly flat

The engine, the store, the compute. As load grows, this barely moves. The free tier starts at zero, no card.

INFERENCE

Scales with work

Model tokens, one draft at a time. This is the line that grows — and the one the batching and memoization layers keep honest.

Memoization means you never pay twice for the same thought. That is a cost lever, not just a reliability one.

ONE INVARIANT TO REMEMBER

The agent never imports the nervous system.

That one boundary is what makes the whole thing portable. The nervous-system *pattern* is invariant; the platform under it is a choice you can change.

Inngest *or* Temporal Restate Dapr *— swap-ready, agent untouched*

PRODUCTION LANGUAGE, NOT CURRICULUM BRANDING

Two founding engineers reached for the same picture on their own — they call it *“the nervous system”* of their product.

DAY AI · A CRM FOR AI-NATIVE COMPANIES, RUNNING ON EVERY PRIMITIVE IN THIS COURSE

Senses wake it.

Reflexes carry it through failure.

Balance keeps it healthy under load.

And a gate lets a human decide.

Add this to your agent, and you cross the line from something you operate to something that operates on its own.

HOW YOU ACTUALLY GET GOOD AT THIS

Reading this won’t make you good. *Building it will.*

Build the worker. Feel the friction as you wrap it. Let each piece of friction teach you which layer it belongs to. Your companion agent writes the code — your job is to direct it, and to watch the nervous system come alive.