KEYSTONE AIC

BUILD IT YOURSELF

The Applied AI Field Guide

How to build AI systems your business runs through — applied, auditable, and owned. The whole method, given away.

You expected a pitch. This is the method instead. If you are serious about systems, you can do this yourself — and this tells you exactly how.

William Ulmer · Applied AI Systems · Owner, Architect
info@keystoneaic.com

START HERE

Why this is free

Everyone serious about their business, their processes, and their quality can already do most of what follows. The tools are public. The techniques are learnable. So there is no reason to lock this behind a form or a sales call, and every reason to hand it over.

What this guide contains is not a secret. It is a discipline. It is the difference between wiring a language model into a folder of scripts and hoping, versus engineering a system that a business can actually run through — one that holds up when the inputs get ugly, that can be audited when someone asks why, and that stays yours after the person who built it moves on.

The philosophy underneath every page comes from two places: industrial and systems engineering, where you learn that design and execution are two different disciplines and the whole art lives in the handoff between them; and a nuclear reactor plant, where you learn that a powerful engine does the work while a human holds the authority and answers for the outcome. That is the entire thesis. Applied AI, not agentic AI. Let the machine carry the load. Never let it carry the responsibility.

> **THE ONE RULE THAT GOVERNS THE REST**
>
> The language model is the most capable workhorse you will ever hire and the least accountable. Your job is not to trust it. Your job is to build the structure that makes trusting it unnecessary.

Read it, build from it, and if you would rather not spend your weekends on it, the address is on the cover.

PART ONE

The engineering comes first

Before a single prompt is written, decide what you are building and what “right” looks like. This sounds obvious and is almost never done. Most AI projects begin with someone typing into a chat window and reacting to whatever comes back. That is tinkering, not engineering, and it does not survive contact with a real operation.

Separate the judgment from the labor

Every system you build has two halves that must never be confused. The judgment half is what a human decides: what the system is for, what counts as a correct output, what must never happen, and who is accountable. The labor half is what the machine does: the parsing, the extraction, the reconciling, the drafting, the relentless high-volume work no person should be doing by hand.

The failure mode of the entire industry is letting the labor half quietly absorb the judgment half. An “agent” told to go accomplish a goal is making judgment calls you never specified, cannot see, and cannot audit. That is the arrangement a systems engineer refuses. You define the judgment explicitly, up front, and you let the machine execute only inside those lines.

Define. Decide. Do.

The sequence is not negotiable and the order matters.

- Define what the system is for. Write the actual purpose in one or two plain sentences. If you cannot state it plainly, you are not ready to build it.
- Decide what right looks like before you build. This is where the rubric is born, and it is the most skipped and most important step in the whole method. You must be able to describe a good output and a bad output in concrete, checkable terms while the system is still on paper.
- Do it. Only now do you open the prompt. Build the smallest working version that produces the defined output, judged against the decided standard.

> **WHY ORDER IS EVERYTHING**
>
> If you build before you decide what right looks like, you will accept whatever the model gives you, because you will have no standard to reject it against. The rubric written after the fact always conforms to the output. The rubric written before the fact governs it.

PART TWO

Rubrics: how you contain the drift

This is the core of the entire method, so it gets the most room. A language model does not fail loudly. It fails by drifting — producing output that is fluent, confident, plausible, and subtly wrong, in ways that compound quietly until someone downstream makes a decision on bad information. You cannot prevent drift. The model is probabilistic by nature. What you can do is contain it, and the instrument of containment is the rubric.

What a rubric actually is

A rubric is an explicit, written standard that defines what a correct output looks like, broken into checkable dimensions, decided before the work is done. It is the specification the system is measured against on every single run. In systems-engineering terms it is the acceptance criteria. In plain terms it is the answer to “how would I know if this came back wrong?” written down before you could be fooled by a confident answer.

A rubric is not a prompt. The prompt tells the model what to do. The rubric is the separate, independent standard you judge the result against — ideally applied by something other than the model that produced the work. When you let the same engine both do the work and grade the work, you have learned nothing. Judgment and execution must stay separate here too.

A rubric has dimensions, not a single score

Do not grade an output as simply good or bad. Break the standard into the specific dimensions that matter for that system, and require each to be met. For a lead-cleaning system the dimensions might be: every record has a valid, dialable phone number; no duplicates survive; each record carries the tier that determines call order; nothing was invented that was not in the source. A record that fails any one dimension fails, no matter how good it looks.

> **THE ANTI-HALLUCINATION DIMENSION**
>
> Every rubric for a system that touches real data must include one rule above all others: nothing in the output may exist that was not supported by the input. This single checkable dimension catches the most dangerous failure a language model produces — the confident invention of facts that were never there.

Rubric rules that hold the line

Over many builds, a small set of enforcement rules earn their place in nearly every rubric. Adopt them as defaults.

- The artifact is the truth. Judge only what the system actually produced, not what it claimed it did. A model that says “I have verified all records” has verified nothing. Grade the records.
- Silence is failure. If the system was supposed to produce something and did not, that is a failed run, not an empty success. Missing output is the most common way drift hides.
- A near-miss is a miss. Partial compliance on a dimension that was defined as required is a failure. “Mostly deduplicated” is not deduplicated.
- Grade against the standard, not against the last run. The bar is what you decided right looks like, not whether today's output is better than yesterday's.

Where the rubric lives in the system

The rubric is not a document you read once and forget. It becomes a gate in the pipeline. The model produces an output, the rubric is applied as an explicit checking step, and only output that passes every required dimension moves forward. Output that fails is caught, logged, and either corrected or held. This is the difference between a system that fails safely and one that fails silently into your CRM.

> **THE RUBRIC IS WHAT MAKES IT AUDITABLE**
>
> Because the standard was written down before the work and applied as an explicit step, you can always answer the question a serious business will eventually ask: how do you know this is right? You point at the rubric and the pass record. That is auditability, and it is what separates a system a business can run through from a clever trick nobody can stand behind.

PART THREE

Test small, correct big

With a rubric in hand you can finally build, and the way you build is deliberately, in small controlled passes, never all at once against your real data.

Run it on a little first

Take a small, representative slice of the real work — twenty records, not twenty thousand — and run the system against it. Small enough that you can inspect every output by hand and compare it to the rubric yourself. This is the prototype station, and its only job is to prove the system produces defined output at the decided standard before you ever point it at volume.

When it is wrong, fix the structure, not the symptom

It will be wrong. That is expected, not alarming. The amateur move is to patch the one bad output by hand and move on. The engineer's move is to ask why the structure allowed that failure, and fix the structure so the whole class of failure cannot recur. One wrong record is a data point. The design flaw behind it is the actual problem. Correct big: fix the cause, then re-run the whole small batch and confirm the fix held without breaking anything else.

> **SMALL MISTAKES ARE CHEAP. SCALED MISTAKES ARE NOT.**
>
> Every failure you find at twenty records costs you minutes. The same failure discovered at scale, after the bad output has flowed into decisions and downstream systems, can cost the trust of a client or the integrity of a database. Never skip the small pass to save time.

Assume drift, not accuracy

Build every system on the assumption that the engine will wander, because it will. This is not pessimism; it is the correct engineering posture toward a probabilistic component. Concretely: never assume an output is right because it reads right; build the checks that catch drift as a permanent part of the system, not a one-time test; and re-verify on a schedule, because a prompt and model that behaved last month can behave differently after an update you did not control.

A system built assuming accuracy breaks the first time the model surprises it, and does so silently. A system built assuming drift catches the surprise, logs it, and holds the line. Same model, same task — the difference is entirely in the discipline around it.

PART FOUR

Build a system, not a folder of scripts

Here is where most “AI tools” reveal what they actually are: a script in a folder on one person's machine, run by hand, remembered by no one, secured by nothing. That is fine for a personal experiment. It is not a system a business can run through, and the gap between the two is architecture.

Persistence: it has to remember

A real system has a place where inputs, outputs, and results live — a database, not a pile of files. This is what lets it be used more than once, by more than one person, and reported on afterward. State that persists is the first thing that separates a system from a stunt.

Access: login, not the honor system

If more than one person will ever touch it, it needs real authentication — accounts, credentials, and permissions — not a shared link and a hope. A tool with no login has no notion of who did what, cannot restrict who sees which data, and cannot be shut off for one user without breaking it for all. Login is not a feature you add later. It is a structural decision you make at the start, because retrofitting identity onto a system that never had it is a rebuild, not a patch.

> **WHY LOGIN IS A SYSTEMS-ENGINEERING DECISION, NOT A CONVENIENCE**
>
> Authentication is what makes a system accountable. It ties every action to a person, every data view to a permission, and every change to a record. Without it you cannot audit, you cannot secure, and you cannot scale past the one person who built it. The same principle as a reactor watch bill, where every action is signed for by name.

What each layer is for

Think of any real system you build as a small stack of honest layers, each with one job.

- The data layer holds the truth: the inputs, the outputs, the pass and fail records. It persists. It is the thing the business actually owns.
- The logic layer does the deterministic work: the parsing, the staging, the rule enforcement, the rubric checks. Ordinary, testable, predictable code — and as much of the system as possible should live here, where behavior is certain.
- The intelligence layer is the model, and it is deliberately the smallest and most tightly bounded layer, called only for the specific heavy lifting that genuinely needs it, always inside the rubric's gate.

- The access layer is who is allowed in and what they can see and do. Login, roles, permissions. It wraps everything else.

> **THE TEST FOR EVERY FEATURE**
>
> Before you let the model do anything, ask: does this need judgment, or just labor? If it needs judgment, it belongs in the logic layer as an explicit rule, or with the human. Only genuine labor goes to the intelligence layer. Give the model the least it can do well, and no more.

PART FIVE

Type it in plainly

With the discipline, the rubric, and the architecture in place, the prompt itself is almost anticlimactic — which is the point. The prompt is the least important part of a well-engineered system, not the most. If your system depends on a magic incantation, the structure around it is too weak.

Say what you actually want

Tell the model plainly what it is doing, in your own words, with the specifics that matter. Give it the input, the exact shape of the output you want back, and the constraints it must honor. Do not perform elaborate prompt rituals; state the task clearly, show one good example if the format is particular, and let the rubric — not clever wording — be what guarantees quality.

```
# state the job plainly, then constrain the output
You are cleaning a raw lead export.
For each row, return: company, valid phone, tier (1-3).

Rules:
- Use ONLY data present in the input. Invent nothing.
- Drop rows with no dialable phone number.
- If a field is missing, leave it blank. Do not guess.
- Return valid JSON only. No commentary.
```

Notice that the prompt already carries pieces of the rubric — invent nothing, do not guess, exact output shape. That is intentional. The prompt asks for the right thing; the rubric independently verifies you got it. Judgment and execution kept separate even here.

The tools exist. They work. Have patience.

None of this requires anything exotic. The models are public, the databases are ordinary, the authentication is standard. What the work actually demands is patience: the willingness to define before building, to write the rubric before you are allowed to be impressed, to test small and fix causes, and to build the boring layers that make a system trustworthy. That patience is the whole secret. It is more discipline than genius, and it is entirely learnable.

THE WHOLE THING

In one page

If you remember nothing else, remember this sequence, because it is the entire method compressed.

- Separate judgment from labor. You decide; the machine does. Never let the machine quietly take over the deciding.
- Define, decide, do — in that order. Write the purpose. Write the rubric. Then build. Never build first.
- The rubric contains the drift. Explicit, dimensional, decided in advance, applied as a gate, enforced by something other than the model that did the work. Nothing invented that was not in the input.
- Test small, correct big. Prove it on twenty before twenty thousand. Fix the structure behind a failure, not the symptom in front of it.
- Assume drift, not accuracy. The engine wanders. Build the checks that catch it, permanently, and re-verify on a schedule.
- Build a system, not a folder of scripts. Persistence so it remembers. Login so it is accountable. Layers so each part has one honest job. The model is the smallest layer.
- Type it in plainly, and have patience. Clear tasks, one good example, and the rubric doing the guaranteeing. The tools work. The discipline is the hard part, and it is learnable.

THAT'S IT. GO.

No trick and no withheld step. This is genuinely how it is done. Most people who read this will not build it — not because they cannot, but because they would rather run their business than build their pipeline. If that is you, the address is below, and the only thing you are buying is the judgment and the years of doing it wrong so you do not have to.

info@keystoneaic.com

Keystone AI Company, LLC · dba Keystone AIC · Applied AI, not agentic