HDCRS SUMMER SCHOOL 2026 · GEOAI AGENT HANDS-ON

Agents for Earth Science

Tools, harnesses, guardrails, and the CARE design process

Muthukumaran Ramasubramanian · NASA IMPACT AI

github.com/NASA-IMPACT/HDCRS-school-2026

About me

Muthukumaran Ramasubramanian — NASA IMPACT AI, The University of Alabama in Huntsville · NASA Marshall Space Flight Center. I lead **AKD (Accelerated Knowledge Discovery)**, the agent development team behind the **AI Agents for Science** initiative.

What the team builds

- **Science agents designed with CARE** — the methodology covered in this lecture; specialized agents for tasks where general chatbots lack the tools, databases, terminology, and ontologies that scientific work requires.
- **Domain agents across NASA science** — ten specialized agents to date, including Earthdata/CMR search, Planetary Data System (PDS) search, Earth-science code search, and science gap analysis.
- **Multi-agent workflows** — end-to-end research pipelines, e.g. a climate modeling workflow.
- **Science-specific guardrails** — factual-integrity and risk evaluation, beyond generic safety filtering.

How the team collaborates

- **SMEs, developers, and helper agents** — the three-party CARE model in daily practice: NASA scientists provide domain knowledge; the team turns it into reviewable agent designs.
- **IBM Research** — guardrails and AI risk frameworks.
- **Development Seed** — engineering.
- **NASA programs** — Worldview, Physical Sciences Informatics, and the Science Discovery Engine as integration targets and data providers.
- **Open community** — designs and code developed in the open with external contributions.

Project page

nasa-impact.github.io/AI-Agents-for-Science

This session

1. **What is an agent** — definition and the agent loop
2. **Tools** — typed functions; the Model Context Protocol
3. **The harness** — the deterministic code around the model
4. **Context engineering** — artifacts instead of ad-hoc prompts
5. **Guardrails** — domain-specific risk evaluation and enforcement
6. **CARE** — a design process for science agents
7. **Hands-on** — the GeoAI agent notebook, top to bottom

Goal of the lecture

Give you the concepts you will see, one by one, in the hands-on notebook — so that when you run `geo_agent.ipynb`, each section maps to an idea introduced here.

Running example

A geospatial event-analysis agent built on Prithvi-EO-2.0: flood detection, burn-scar mapping, and crop-type classification from natural-language requests.

What is an AI agent?

A specialized AI agent is **an LLM connected to specific tools, given context, and designed to perform a specialized task**.

	CHATBOT	AGENT
Acts by	generating text from its parameters	calling tools and observing the results
Unit of work	one question, one answer	a multi-step task run to completion
Knowledge	frozen at training time	retrieved from data systems at run time
Verification	reader's judgement	tool outputs, logs, provenance records

Why this matters for Earth observation

Answering "map the flooding near Houston in late August 2017" requires geocoding, an imagery-archive search, a model inference job, and statistics over a raster — none of which a language model can do by generating text alone.

The agent loop

Prompt assembly
system prompt + history + memory

→

LLM call
model proposes next step

→

Parse
text answer or tool call?

→

Tool execution
run the function, capture result

→

Append result
back into context, repeat

- The loop itself is ordinary, deterministic code — a `while` loop. The model only decides *which step to take next*.
- The loop exits when the model produces a final answer instead of a tool call, or when a limit (steps, tokens, cost) is reached.
- Everything around the LLM call — assembly, parsing, execution, limits, state — is infrastructure we control. This is the **harness**, covered shortly.

Tools

- A tool is a **typed function with a description**, exposed to the model: name, input schema, output schema, documentation.
- The description is read by the LLM, not by humans — it must be precise about inputs, outputs, and edge cases. **Tool descriptions are documentation for the model.**
- Good tools return **structured, compact output** (selected fields, not raw dumps) so results stay useful inside a finite context window.

```
# tools/geocode/geocode_location.py — the schema IS the contract
class GeoCodeLocationInput(InputSchema):
    """Place name or description to geocode."""
    query: str = Field(..., description=
        "Place name, event/location description, or bbox "
        "coordinates as user-provided text")

class GeoCodeLocationOutput(OutputSchema):
    bbox: list[float] | None = None        # [W, S, E, N]
    display_name: str | None = None
    candidates: list[dict] | None = None   # if ambiguous
```

Notebook §2.3 prints every registered tool with its description — inspect them before the agent runs anything.

TUTORIAL TOOL	PURPOSE
`geocode_location`	place name → bounding box
`check_hls_availability`	HLS imagery search for AOI/date; cloud-cover screening

`query_active_fires`
`query_fire_history`
`query_surface_water`
`query_crop_landcover`
	auxiliary dataset signals, used to infer the task type when the user does not state it

`run_prithvi_inference`	Prithvi-EO-2.0 inference → GeoTIFF mask + predictions

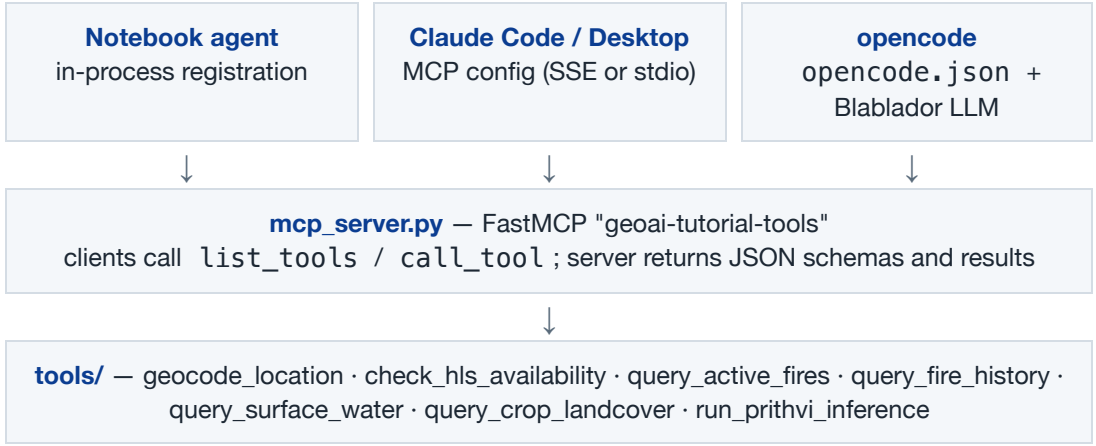
The Model Context Protocol (MCP)

- The **Model Context Protocol (MCP)** is an open standard for exposing tools to LLM applications over a common interface.
- Instead of binding tools to one agent framework, you run a **tool server**; any MCP-aware client can discover and call the tools.
- In this tutorial, `mcp_server.py` serves all seven runtime tools locally.

```
# mcp_server.py — the entire server, abridged
mcp = FastMCP("geoai-tutorial-tools",
    instructions="Geospatial tools for the GeoAI agent: "
        "geocoding, HLS availability, Prithvi inference, ...")
for tool in TOOLS:          # the same tool objects the
    register_mcp_tool(      # notebook agent registers
        tool_converter(tool), mcp)
mcp.run(transport="sse", host="127.0.0.1", port=8081)
```

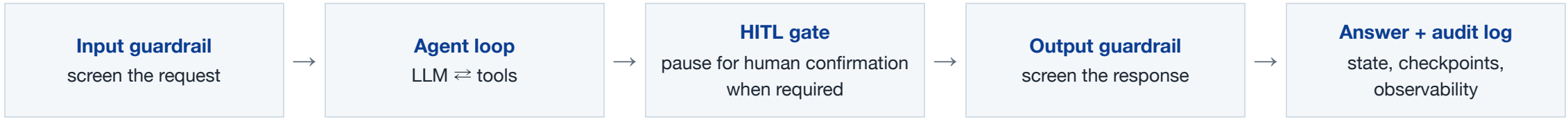
```
# client side — .claude/settings.json (Claude Code)
{ "mcpServers": { "geoai-tools": {
    "type": "sse", "url": "http://127.0.0.1:8081/sse" } } }
```

How the pieces connect



The protocol decouples tool implementations from the choice of model and agent framework: the same tool definitions serve all three clients without modification.

The harness: deterministic code around the model



- Most of a production agent system is **deterministic infrastructure**, not model calls — prompt assembly, parsing, tool execution, retries, state, logging.
- In practice, overall system reliability is determined largely by the harness — tool quality, error handling, state management — rather than by the choice of model alone.
- Human-in-the-loop (HITL) is a harness feature: the run pauses, state is checkpointed, and it resumes with the human's answer.
- For science, the harness is also where **provenance** is produced — every tool call and result is recorded, independent of the model.
- To the harness, the LLM is just another callable component — tools and model alike are typed, invocable resources the deterministic loop coordinates.

From prompt engineering to context engineering

"Context engineering is the delicate art and science of filling the context window with just the right information for the next step." — A. Karpathy, 2025

Four operations

Write	externalize state outside the window: files, scratchpads, decision logs
Select	pull in only what is relevant: retrieval, manifests, on-demand reads
Compress	summarize as context grows: compaction, structured tool outputs
Isolate	partition concerns: sub-agents, schema boundaries, fresh sessions

How contexts fail

Poisoning	a bad tool output or hallucination enters context and propagates as fact
Distraction	so much history that the model loses the actual question
Confusion	irrelevant outputs and stray instructions drown the signal
Clash	two pieces of context disagree; the model picks one — sometimes wrong

Measured "context rot": model quality degrades as input grows. Treat the window as a finite attention budget, not free storage.

Artifact-driven context: agent behavior specified as files

```
artifacts/<agent>/
├── scope.md         purpose · users · workflow · success
├── contexts/        domain knowledge, conventions
│   ├── index.md     manifest — what's here, when to read it
│   └── <topic>.md
├── tools/           tool inventory + per-tool notes
├── output.md        output format
├── reasoning.md     reasoning strategy, stop conditions
├── guardrails/      safety rules, enforcement policy
└── agents.md        ENTRY POINT — the system prompt
```

- `agents.md` is loaded verbatim as the system prompt; it references the other files by relative path.
- The agent reads the rest **on demand** through read-only filesystem tools (`ls` , `read_file` , `glob` , `grep`) — nothing is stitched together in code.
- `index.md` manifests enable targeted reads instead of loading whole directories.
- Files are **diff-able, reviewable, versioned**, and swappable across models — a domain expert can review agent behavior without reading Python.
- This is the same shape used by Anthropic's Skills spec and by NASA IMPACT AI's production agents (AKD).

Guardrails: generic safety filters are insufficient for science

An agent can be polite and non-toxic while still hallucinating physical constants, citing papers that do not exist, or reporting results with the wrong units. Science guardrails enforce **domain-specific integrity**, not just harmlessness.

Input guardrail — screen the request

prompt-injection	attempts to override system instructions
jailbreaking	attempts to bypass safety constraints
harmful-content	requests for harmful use (targeting, surveillance)
toxic-language	abusive or discriminatory content

Output guardrail — screen the response

hallucination	fabricated facts, statistics, or citations
missing-attribution	claims without sources or tool-backed provenance
overgeneralization	sweeping claims beyond what the data shows
false-certainty	overconfident statements without uncertainty

Implementation in the notebook (§3): an LLM evaluator with structured output scores text against each risk category — one verdict per category, programmatically checkable.

Domain-specific guardrails for Earth observation

Geospatial risk categories (notebook §3.2)

Coordinate plausibility	do the coordinates fall on land, inside the stated region? (e.g. crop classification 15 km offshore)
Sensor operational period	does the date range fall within the sensor's lifetime? (e.g. Sentinel-2 data "from 2011")
Resolution–task compatibility	is 30 m imagery appropriate for sub-hectare field mapping?

Adding a custom guardrail = defining new risk categories and composing them with the existing list. No framework changes.

Enforcement policy (from the artifact)

SIGNAL	ACTION
harmful input category	**refuse**
hallucination detected in output	**rewrite** — remove unsupported claims; refuse if not possible
jailbreak content in output	**refuse**
out-of-distribution request	**clarify** — ask the user to re-scope to flood / burn / crop

From the Prithvi artifact's SME-validated enforcement matrix (`guardrails/safety_and_guardrails.md`).

The guardrail stack: four layers

A guardrail is a **separate system** that evaluates an agent's input and output against defined risk criteria and decides to block, flag, or log — an independent evaluator that checks the agent, not part of it. No single guardrail catches everything: safety, scientific quality, factual consistency, and physical verification are different problems, addressed by different layers.

LAYER	ORIGIN	STAGE	WHAT IT DOES
Granite Guardian	IBM	input + output	safety classifier — jailbreaks, toxicity, harmful content; one pre-trained model, one forward pass: fast and cheap
Risk Agent	IBM	input + output	scientific-quality judge — hallucination, attribution, overgeneralization, uncertainty, consistency; generates custom criteria per response and rolls their verdicts up into one weighted score
FactReasoner	IBM	post-generation	factual consistency — compares the output against retrieved evidence; probability score per claim; handles conflicting evidence rather than majority-voting
GeoGuard	AKD	post-generation	science verification — decomposes claims and checks them against real data sources (NOAA, USGS, NASA MODIS, OSM); verdicts come from measurements, not model judgment

Risk Agent asks whether the model considers the output risky; GeoGuard asks whether the weather station agrees. Source: R. Sahoo, *Guardrails for Geo AI Agents*, ESA–NASA Workshop 2026.

Risk Agent: a fresh rubric every time

How it works

- **1 · Criteria generation** — for each risk category, the LLM reads the output and its context and generates 3–8 specific criteria, each tagged high / medium / low importance.
- **2 · Scoring tree** — each criterion gets a true/false verdict; each category aggregates its criteria (any high-importance failure fails the whole risk); a weighted overall score from 0 to 1.
- **3 · Evaluation** — the LLM judges each criterion against the actual output.
- **4 · Risk report** — for failed risks, a prose explanation citing which criteria failed and why.

Why independent, dynamic, and hierarchically scored

Independent — a separate system with no shared prompt, context, or state: what fools the generator does not automatically fool the judge, and agents never grade their own work. **Dynamic** — a fixed rubric cannot anticipate every output; per-response criteria match the claims actually made and cannot be gamed in advance. **Hierarchical scoring** — each criterion gets its own verdict, criteria roll up into risk categories, and categories combine into one weighted score: you can trace exactly which check failed and tune its weight, instead of accepting one opaque overall judgment.

Example: the scoring tree on a Hurricane Harvey response

Output under review: "Hurricane Harvey caused catastrophic flooding across **the entire Texas coastline**, with **every county** experiencing severe damage. The storm produced **unprecedented** rainfall that **permanently** altered the coast."

RISK CATEGORY	GENERATED CRITERIA (IMPORTANCE → VERDICT)	RESULT
Overgeneralization	"entire Texas coastline" (high → false) · "every county" (high → false) · "permanently altered" (medium → false)	**fail**
Uncertainty	"unprecedented" (high → false) · no uncertainty language present (medium → false)	**fail**
Hallucination	core event — Harvey flooding in Texas (high → true) · rainfall claim broadly supported (medium → true)	**pass**

Weighted roll-up: overall score **0.4** against a 0.9 threshold → **fail**, with the report: "Response overgeneralizes flood impact to the entire coastline. Uses absolute language without evidence or uncertainty markers."

Works on input or output; the rubric is generated fresh for every response. The notebook §3 evaluator is a minimal version of this pattern. Source: R. Sahoo, *Guardrails for Geo AI Agents*.

FactReasoner: probabilistic factuality assessment

Pipeline

IBM Research (arXiv:2502.18573, EMNLP Findings 2025).

LLM response — long-form answer to assess

↓

1 · Atomizer — split into atomic claims, each independently verifiable

↓

2 · Reviser — decontextualize: resolve pronouns, names, implicit references

↓

3 · Context retriever — evidence per claim from external knowledge sources

↓

4 · Relation assessment — entailment / contradiction between claims and contexts

↓

5 · Probabilistic reasoning — graphical model over claims + evidence → posterior P(claim supported), aggregate factuality score

Example: per-claim posteriors

ATOMIC CLAIM	RETRIEVED EVIDENCE	P(SUPPORTED)
"Harvey made landfall on August 25, 2017"	entailed by multiple independent sources	high (≈0.99)
"Harvey was a Category 4 hurricane at landfall"	entailed	high (≈0.97)
"Harvey produced 75 inches of rain in Houston"	**contradicted** — the recorded storm-total maximum is 60.58 in (Nederland, TX)	low (≈0.05)
"Harvey is the costliest U.S. hurricane on record"	**conflicting** — sources tie Harvey with Katrina	intermediate (≈0.5)

The intermediate posterior in the last row expresses genuine evidential conflict — a binary judge would have to round it away. Probabilities illustrative.

Why probabilistic, not another LLM vote

Calibrated probabilities instead of binary verdicts; aggregation of multiple — possibly conflicting — evidence sources; joint reasoning enforces consistency across claims.

GeoGuard: measurement-based verification for the Prithvi agent

How it works — purpose-built for geospatial AI

- **1 · Claim extraction** — one LLM call decomposes the output into atomic, decontextualized claims, grouped by event type (flood, storm, …).
- **2 · Tool selection** — for each claim, the LLM picks verification tools by the claim's signature: what data would you need to check this?
- **3 · Verification** — an agent loop calls the tools and gets real measurements back. The verifier never sees the original input, only the claim and the tool results — preventing anchoring bias.
- **4 · Rubric** — a holistic confidence score per claim, aggregating the yes/no checks; no new tool calls, only reasoning over collected evidence.
- **5 · Final report** — per-claim verdicts (**supports / contradicts / inconclusive**), tool-call traces, and an overall confidence score.

Example walkthrough

Agent output: "Hurricane Beryl made landfall near Matagorda Bay… sustained winds of 80 mph… widespread flooding in Houston… over 100 mm of rain in 24 hours."

EXTRACTED CLAIM	VERDICT FROM DATA
sustained winds of 80 mph	**contradicts** — station measurements disagree
significant storm surge	**supports**
widespread flooding in Houston	**supports**
over 100 mm rain in 24 h	**supports**

Overall verdict: **contradicts** — the response is flagged despite most claims holding, because verification is against NOAA / USGS / NASA MODIS / OSM data, not against model opinion.

In the Prithvi agent, GeoGuard runs post-generation alongside the science output guardrail; the notebook §3.2 geo risk categories are a lightweight, LLM-judged version of these checks.

Requirements for agents in scientific use

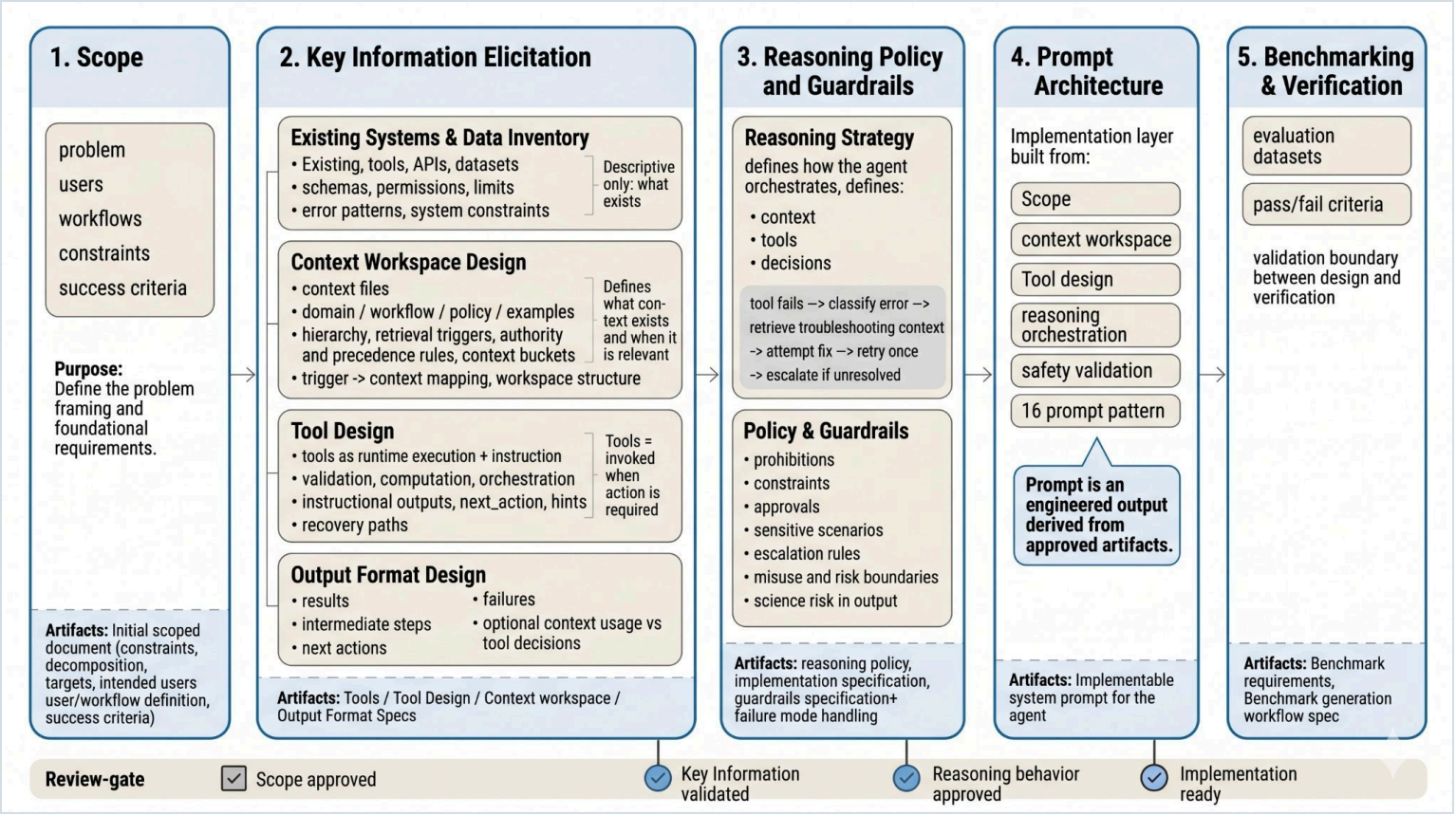
- **Provenance.** Every result traceable to tool calls: which imagery, which dates, which model run. The Prithvi agent emits a JSON audit log alongside its narrative answer.
- **No fabrication.** Never claim an inference ran unless it succeeded; never invent output links or statistics.
- **Bounded interpretation.** Report detections and statistics; do not draw scientific conclusions beyond what the model output shows.
- **Disclosed degradation.** If a nearby date was substituted or cloud cover is high, say so plainly.
- **Human-controlled decisions.** Some choices must stay with the user: confirming the resolved location and dates before inference, accepting degraded data, interpreting results.
- **Scope discipline.** The agent refuses anything outside its three supported tasks and explains why — a narrow, well-defined agent is more trustworthy than a broad one.
- **Credential hygiene.** Never echo tokens or Earthdata credentials in chat or logs.

The common thread

These are not model properties. They are **engineered** — in the artifact, the guardrails, and the harness.

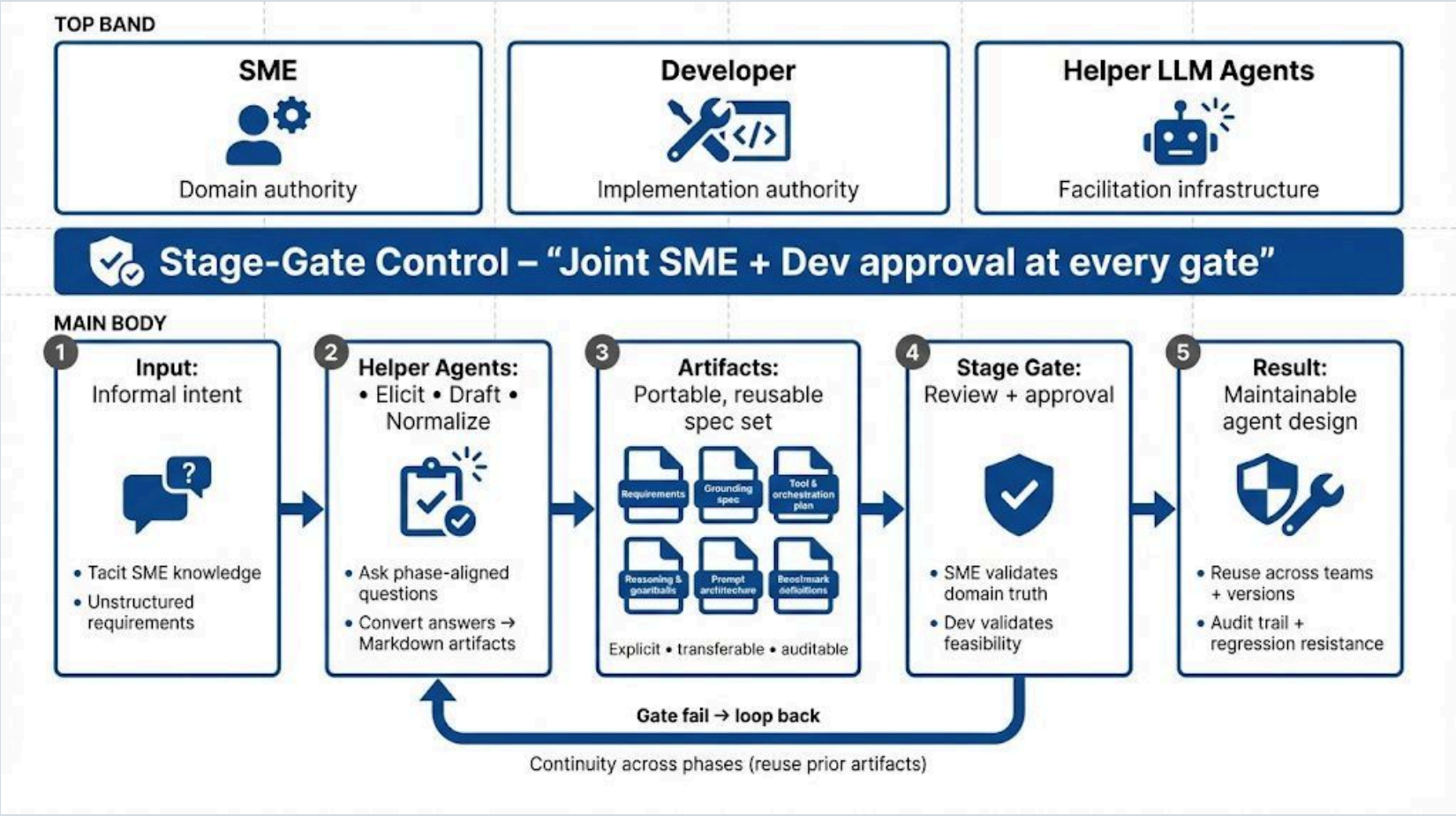
CARE: Collaborative Agent Reasoning Engineering

A NASA IMPACT AI design process that moves agent building from ad-hoc prompting to disciplined engineering. Goal: **make expert reasoning explicit, reviewable, and verifiable.** Each stage produces reviewable artifacts and passes a review gate before the next begins.



Source: NASA Technical Memo on CARE. The hands-on covers stages 1–4; stage 5 (benchmarking & verification) is out of scope for this session. Stage outputs map directly to the artifact files: `scope.md` → `contexts/` `tools/` `output.md` → `reasoning.md` `guardrails/` → `agents.md`.

CARE: three-party collaboration with stage gates

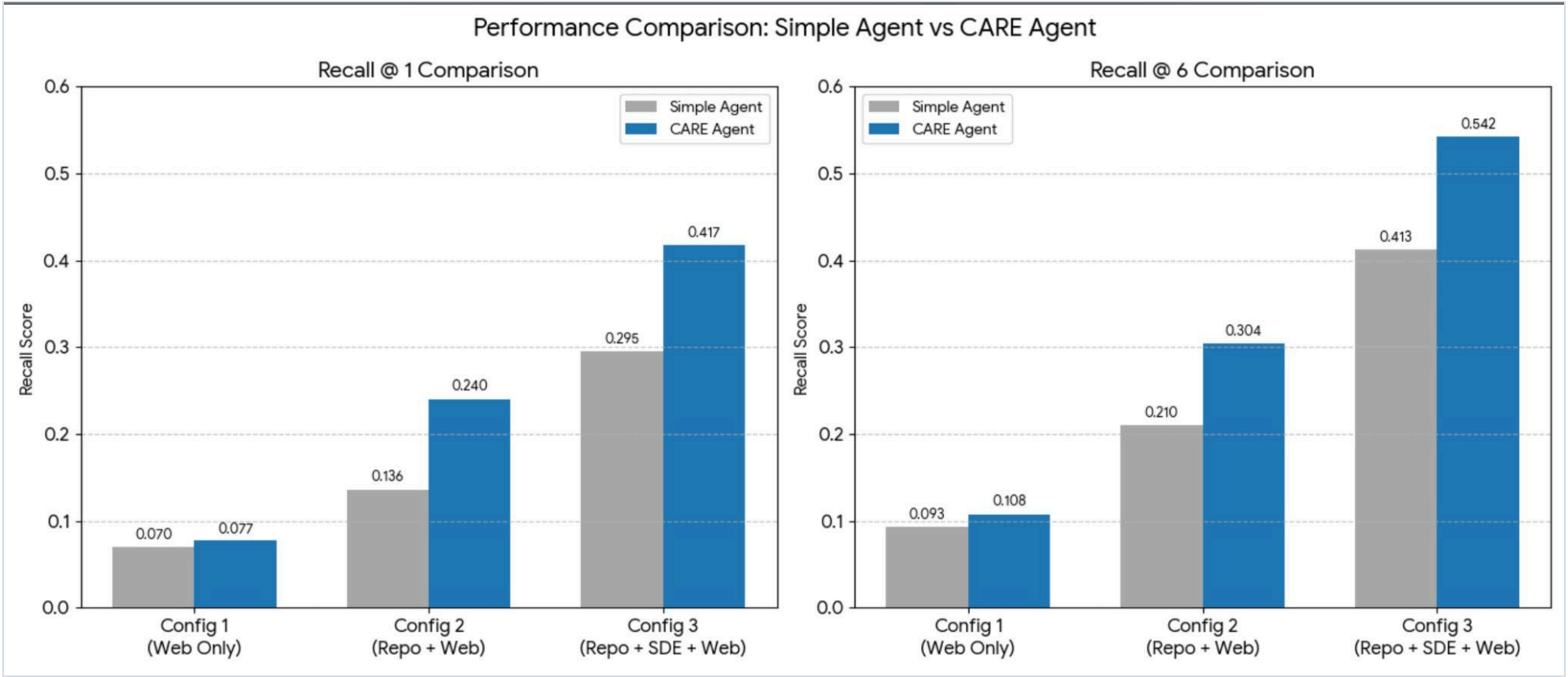


Three parties: the subject-matter expert holds domain authority, the developer holds implementation authority, and **helper agents** (the CARE interviewer) translate informal intent into structured artifacts.

Stage-gate control: joint SME + developer approval at every gate; artifacts are portable and reusable; a gate can fail and loop back. The result is a maintainable agent design with an audit trail.

Case study: CARE-designed agent vs ad-hoc baseline

Earth-science code search agent — a CARE-designed agent compared against a simple (ad-hoc prompted) agent using the **same model and same tools**, across three tool configurations.



The CARE-designed agent achieves higher recall (Recall@1 and Recall@6) in every configuration — the gap widens as more tools are available. The design process, not the model, made the difference. Source: NASA Technical Memo on CARE.

Why CARE — and when to use it

Benefits of the process

- **Tacit knowledge is captured.** Expert reasoning that would otherwise live in an SME's head — edge cases, failure modes, refusal boundaries — is elicited and written down where it can be reviewed and corrected.
- **Behavior is auditable.** Stage gates produce an approval trail; artifacts are diff-able, so behavior changes are visible and reviewable like code.
- **Artifacts outlive models.** The design is plain markdown, independent of any framework or model — swap the model without redesigning the agent.
- **Fewer tokens per run.** The artifact encodes which tool to call, when, and in what order, so the agent does not spend turns probing tools to discover the right pathway — fewer tool calls, lower cost and latency.
- **Measured performance gains** with the same model and tools (previous slide).

When CARE is the right tool

- The task is **specialized and recurring** — the agent will be run, maintained, and reused, so design effort amortizes.
- **Correctness and provenance matter** — results feed scientific or operational decisions.
- The required knowledge is **tacit and held by SMEs** who are not agent developers.
- Multiple parties must **agree on and review** the agent's behavior.

When it is not

- One-off exploratory questions — just use a chat model.
- Tasks fully covered by existing documentation a generic agent can already read.
- Prototyping to discover whether a task is feasible at all — prototype freely first, formalize with CARE once it is.

Codified knowledge permits smaller models

The more knowledge the artifact makes explicit, the less must come from the model's implicit reasoning — **smaller, more efficient models** become sufficient, lowering cost and enabling local deployment.

The hands-on: `geo_agent.ipynb`

Part 1 — design an agent (CARE)

Chat with the CARE interviewer agent. It walks you through the four phases and writes the artifact files into a workspace as you answer — you experience the design process from the SME's seat.

Part 2 — run a designed agent (Prithvi)

Wire the pre-built Prithvi artifact into a runnable agent and exercise it end to end.

NOTEBOOK SECTION	CONCEPT FROM THIS LECTURE
§0 Setup	backends, models, paths
§1 Context engineering	artifact structure; CARE interviewer
§2 Building the agent	tools, system prompt from `agents.md`
§3 Guardrails	input/output/geo risk evaluators
§4 Putting it together	full agent + chat interface

What the Prithvi agent does

natural-language request
("flood map for Houston, Aug 30 2017")

geocode → bounding box; resolve dates

check HLS imagery availability + cloud cover

run Prithvi-EO-2.0 inference (flood / burn scar / crop)

GeoTIFF map + area statistics + caveats + audit log

Also included: a local MCP server exposing the same tools to Claude Code and opencode — try the same task from a different client.

The Prithvi agent: overview and capabilities

A **geospatial event-analysis assistant**: given a natural-language request, it runs the full pipeline — query → data access → Prithvi-EO-2.0 inference → results. It is deliberately narrow: **flood detection, burn-scar mapping, and crop-type classification**, nothing else. Its entire behavior is defined by a CARE-designed artifact.

Capabilities

- **Feasibility checking** — decides whether a request fits the three tasks; explains the limitation when it does not.
- **Location & date resolution** — geocodes place names to a bounding box; resolves vague dates ("last summer") against event catalogs (storm and fire records); asks when still uncertain.
- **Task inference** — if the task type is unstated, consults dataset signals (active fires, surface water, fire history, crop land cover) to pick flood vs burn vs crop.
- **Imagery verification** — checks HLS availability and cloud cover; falls back to ±3 days for floods/burns; finds three clean, well-spaced dates for crops.
- **Inference & reporting** — runs Prithvi-EO-2.0 (30 m flood/burn masks; 13-class crop map) and returns a GeoTIFF, area statistics, caveats, and a JSON audit log.

Questions it can answer

- "Generate a flood map for Houston, Texas for August 30, 2017."
- "Map the burn scars from the wildfires near Lahaina, Maui, August 2023."
- "Produce a crop-type map for farmland around Lubbock, Texas for the 2023 season."
- "Did the storm last May flood the area around this town? Show the extent." — vague location/date, resolved via geocoding and event catalogs
- "Something happened near this region in August — can you check?" — task type inferred from dataset signals

Questions it refuses, by design

- "Detect deforestation in the Amazon." — outside the three supported tasks; explains scope
- "Track vehicles near this facility." — surveillance/targeting; guardrail refusal
- "Give me step-by-step instructions to rebuild this pipeline." — demo runner; refuses per guardrails

Practical notes

- The session runs in the **JupyterHub environment provided by the organizers** — log in, start a server, open `GeoAI-Agent-Tutorial/geo_agent.ipynb`.
- **Run the cells top to bottom.** The notebook is self-contained; the CARE prompt library is cloned automatically on first run.
- API credentials go in a `.env` file next to the notebook — copy `.env.example` and fill in your keys.
- During the CARE interview, answer as the domain expert — the interviewer elicits; it does not advise.

Things to try once it runs

- In-scope queries: flood detection, burn-scar mapping, crop classification for a place and date you choose.
- **Out-of-scope queries** — confirm the agent refuses per its `guardrails/` rules.
- Adversarial prompts against the guardrail pipeline — watch what gets flagged and why.
- Inspect the audit log and verify the provenance of a result.

Summary

1 · An agent is a system

A model, tools, and context inside a deterministic harness. Reliability depends primarily on the harness, not on the model alone.

2 · Context is engineered

Write the agent's knowledge as reviewable artifact files; load them selectively. The context window is a finite attention budget.

3 · Science guardrails are domain-specific

Provenance, plausibility, attribution, and bounded interpretation — enforced as policy (refuse / rewrite / clarify), not assumed.

4 · Design is a process with gates

CARE makes expert reasoning explicit and SME-reviewed. A well-designed artifact outperforms ad-hoc prompting with the same model and tools — and the more knowledge is codified, the smaller the model that suffices.

AI Agents for Science
nasa-impact.github.io/AI-Agents-for-Science

Tutorial materials
github.com/NASA-IMPACT/HDCRS-school-2026

Materials: github.com/NASA-IMPACT/HDCRS-school-2026 · CARE prompts: github.com/NASA-IMPACT/AKD-CARE · Further reading: Anthropic, *Effective context engineering for AI agents* (2025) · Chroma, *Context Rot* (2025) · NASA Technical Memo on CARE