Intent Engineering for Local AI Agents: A Practical Guide

February 25, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Intent engineering means making your agent's goals, decision boundaries, and value hierarchies machine-readable. Instead of 'be helpful and organize my files,' you define what 'organize' means, when the agent should act vs ask, what to prioritize when speed conflicts with thoroughness, and what it should never do. This article gives you the practical tools: a system prompt evolution from vague to intent-engineered, a Python decision boundary config, a YAML value hierarchy, memory patterns for intent persistence, and a complete starter template you can adapt for any local agent. The theory is in our companion article on the prompt→context→intent progression. This is the how-to.

📚 **Related:** Intent Engineering Theory: The Prompt→Context→Intent Progression · Building AI Agents with Local LLMs · Context Rot and the Forgetting Fix · Session-as-RAG Memory · Function Calling with Local LLMs · Planning Tool

Our companion article covers why intent engineering matters — the Klarna story, the progression from prompt engineering to context engineering to intent engineering, and why 60% of companies see little or no value from AI agents despite massive investment.

This article is the how-to. Practical patterns, code examples, and a starter template for encoding intent into local AI agents.

Why local agents need this more than cloud agents

Cloud API calls are stateless. You send a prompt, get a response, and the connection closes. If the agent makes a bad judgment call, the blast radius is one API response.

Local agents are different. They run for hours or days. They accumulate state. They make hundreds of decisions without human oversight. And because they're running on your hardware — often managing your files, email, or code — a bad judgment call can have real consequences. Summer Yue's OpenClaw email deletion incident happened because a local agent running for an extended session lost a safety instruction during context compaction.

The longer an agent runs, the more it needs encoded intent rather than session-level instructions. Instructions get summarized, compressed, or dropped as context fills up. Intent structures that live outside the context window — in config files, memory systems, or hard-coded decision boundaries — survive compaction.

---

1. Encoding goals your agent can act on

Most agent system prompts look like this:

```
You are a helpful AI assistant. Be thorough, accurate, and friendly.
Organize the user's files when asked. Be careful with deletions.
```

This is a vibe, not a goal. “Helpful” is unmeasurable. “Thorough” conflicts with “efficient” and the agent has no way to resolve the conflict. “Be careful with deletions” is the instruction that got dropped from Summer Yue’s context window.

Here’s the same agent with intent-engineered goals:

```
ROLE: File organization agent for ~/Documents

OBJECTIVE: Reduce retrieval time for frequently accessed files.

SIGNALS:
- Files accessed >3x in past 30 days = high priority
- Files unaccessed for >180 days = archive candidates
- Files >100MB with no access in 90 days = storage review candidates

ACTIONS:
- Move files matching naming patterns to categorized subdirectories
- Create date-based archive folders for old files
- Generate a changelog of every move (what, from, to, why)

CONSTRAINTS:
- NEVER delete files. Move to ~/Archive instead.
- NEVER rename files. Only move between directories.
- NEVER modify files in ~/Documents/Legal or ~/Documents/Tax
- If unsure about categorization, move to ~/Documents/_Review
```

The difference: the second version tells the agent what “organize” means in measurable terms, what actions are available, and where the hard boundaries are. A model can follow this without judgment calls about what “careful” or “thorough” means.

The evolution pattern

Most intent-engineered prompts go through three stages:

Stage 1 (vague): “Help me manage my email inbox.”

Stage 2 (context-rich): “Here are my email rules, my contact list, my calendar, and my priority settings. Manage my inbox.”

Stage 3 (intent-engineered): “Delete obvious spam (confidence >0.95). Archive newsletters older than 7 days. Flag any email from contacts in my CRM that mentions a project deadline. Draft a reply for client emails about active projects — put drafts in my review folder, never send directly. Escalate anything mentioning legal, compliance, or termination to my phone immediately.”

Stage 2 gives the agent information. Stage 3 gives it priorities, thresholds, and boundaries.

---

2. Decision boundaries

The question every agent faces hundreds of times per session: should I act, or should I ask?

Most frameworks leave this to the model’s judgment, which is why agents either do nothing useful (too cautious) or delete your email inbox (too aggressive). The solution is explicit decision boundaries.

Autonomy levels

A framework from researchers at the University of Washington (Feng, McDonald, Zhang — published July 2025) defines five levels of agent autonomy:

Level	Agent role	Your role	Example
L1	Suggests, waits for approval	Operator	“I found 3 spam emails. Delete them?”
L2	Drafts plans, makes minor decisions	Collaborator	Deletes obvious spam, drafts replies for review
L3	Plans and executes independently	Consultant	Manages inbox autonomously, messages you for edge cases
L4	Makes most decisions, asks for high-impact only	Approver	Full inbox management, asks before anything involving money

Level	Agent role	Your role	Example
L5	Fully autonomous	Observer	You only intervene to shut it down

Most local agents should operate at L2 or L3. L4-L5 requires the kind of mature governance that only 21% of organizations have according to Deloitte's 2026 survey.

A practical decision boundary config

```
DECISION_BOUNDARIES = {
    "email_triage": {
        "act_autonomously": [
            "delete_spam",           # confidence > 0.95
            "archive_newsletters",    # older than 7 days
            "label_by_sender",        # known contacts only
        ],
        "act_and_notify": [
            "flag_urgent",            # deadline mentions, escalation language
            "move_to_project_folder", # matched to active project
        ],
        "draft_and_wait": [
            "reply_to_client",        # never send without review
            "forward_to_colleague",   # never forward without review
        ],
        "escalate_immediately": [
            "legal_mention",          # any mention of legal/compliance/lawsuit
            "financial_threshold",    # any amount > $500
            "unknown_sender_attachment", # potential phishing
        ],
        "never_do": [
            "delete_non_spam",        # archive instead
            "send_reply_directly",    # all replies go to draft
            "access_sent_folder",     # read-only for sent mail
            "modify_filters",         # user manages filter rules
        ],
    }
}
```

The agent checks this config before every action. "Can I do this autonomously? Do I need to notify? Do I need to wait for approval? Should I escalate?" These aren't suggestions buried in a system prompt — they're hard boundaries the agent's action loop enforces.

Implementing the check

```
def check_boundary(action_type, category="email_triage"):
    boundaries = DECISION_BOUNDARIES[category]

    if action_type in boundaries["never_do"]:
        return "blocked"
    if action_type in boundaries["escalate_immediately"]:
        return "escalate"
    if action_type in boundaries["draft_and_wait"]:
        return "draft"
    if action_type in boundaries["act_and_notify"]:
        return "act_notify"
    if action_type in boundaries["act_autonomously"]:
        return "act"

    return "draft"  # default to cautious
```

The default is “draft” — if the action isn’t explicitly authorized, the agent drafts it for review rather than executing. This is the opposite of most agent frameworks, where the default is “try it and see.”

---

3. Value hierarchies for tradeoffs

Agents constantly face tradeoffs. Speed vs thoroughness. Precision vs recall. Cost vs quality. Without an explicit hierarchy, the model resolves these based on whatever patterns it learned during training — which may not match your priorities.

A structured value hierarchy

```
# agent-values.yaml
agent:
  name: "research-assistant"

  value_hierarchy:
    # Listed in priority order. When values conflict,
    # higher-ranked values win.
    - accuracy:
        description: "Never fabricate citations or invent data"
        weight: 1.0  # non-negotiable
```

```
    - recency:
        description: "Prefer sources from last 12 months"
        weight: 0.8
        exception: "Foundational papers/docs have no age penalty"

    - thoroughness:
        description: "Cover all relevant angles"
        weight: 0.6
        limit: "Stop at 5 sources per claim unless user asks for more"

    - speed:
        description: "Minimize response latency"
        weight: 0.4
        note: "Always loses to accuracy. If uncertain, take more time."

  conflict_resolution:
    accuracy_vs_speed: "accuracy wins, always"
    recency_vs_thoroughness: "cite the recent source, note older alternatives"
    thoroughness_vs_speed: "default to speed for simple questions, thoroughness for research tas
```

The agent loads this at startup and references it when making tradeoff decisions. “Should I spend another 30 seconds checking a second source, or return what I have?” The value hierarchy says: for a research task, thoroughness beats speed. For a simple question, speed wins. Accuracy always trumps both.

Loading values into agent context

```
import yaml

with open("agent-values.yaml") as f:
    config = yaml.safe_load(f)

values = config["agent"]["value_hierarchy"]
value_prompt = "VALUE HIERARCHY (highest priority first):\n"
for i, value in enumerate(values):
    name = list(value.keys())[0]
    desc = value[name]["description"]
    value_prompt += f"{i+1}. {name}: {desc}\n"

# Inject into system prompt
system_prompt = f"""
{base_prompt}

{value_prompt}
```

```
When values conflict, higher-ranked values always win.
"""
```

This lives in a config file, not in the system prompt itself. If the system prompt gets compacted, the agent can reload values from the file. This is how intent survives context rot.

---

4. Memory as intent persistence

Without persistent memory, intent resets every session. The agent starts fresh, with no history of past decisions, no record of what worked, and no accumulated judgment.

The four memory tiers

Google's Agent Development Kit formalizes this into four layers that map well to local agent architecture:

Tier	What it stores	Lifetime	Local equivalent
Working context	Current task state	One request	System prompt + current conversation
Session state	Conversation goals, user preferences	One session	Session-as-RAG conversation log
Long-term memory	Extracted facts, learned patterns	Persistent	ChromaDB / SQLite with semantic search
Artifacts	Files, outputs, reports	Persistent	Local filesystem with version tracking

For intent engineering, the critical tier is long-term memory. This is where the agent stores not just what happened, but what it learned from what happened.

Episodic memory for decisions

Most memory systems store facts: "User prefers dark mode." "Client X's deadline is March 15." Facts are context engineering — they tell the agent what to know.

Intent-aware memory stores experiences:

```
decision_log = {
    "timestamp": "2026-02-25T14:30:00",
    "situation": "Client email about late delivery, angry tone",
    "options_considered": [
        "Standard apology template",
        "Escalate to human",
        "Offer discount + expedited shipping"
    ],
    "decision": "Escalate to human",
    "rationale": "Angry tone + financial issue = above autonomous threshold",
    "outcome": "Human agent resolved with 15% discount, client satisfied",
    "lesson": "Angry + financial = always escalate. Standard template would have made it worse."
}
```

Over weeks and months, these logs accumulate into a corpus of judgment. The next time the agent faces a similar situation, it can search past decisions: "What did I do last time a client was angry about a financial issue? What worked?"

This is the difference between giving an agent a briefing packet (context) and giving it a history of experience (intent). See our Session-as-RAG guide for the implementation pattern — it works for decision logs the same way it works for conversation history.

How context rot undermines intent

The context rot problem is an intent problem, not just a context problem. When context compaction drops instructions, it drops encoded intent. The agent forgets what it's supposed to care about, not just what you told it.

The fix is storing intent structures outside the context window:

- Decision boundaries in a config file (reloaded on each action)
- Value hierarchies in YAML (referenced, not inlined)
- Decision logs in a searchable database (queried when relevant)
- Hard constraints in the action loop code (not in the prompt at all)

If the only place your agent's intent lives is the system prompt, it will eventually be compacted away. Put the important parts in code and config.

---

5. Feedback loops

85% of organizations expect to customize AI agents for their business. Only 21% have a mature model for agent governance (Deloitte 2026). The gap is feedback loops — measuring whether the agent is doing what you intended, not just whether it's doing something.

Decision logging

The simplest feedback loop: log every decision the agent makes.

```
import json
from datetime import datetime

def log_decision(action, rationale, boundary_check, confidence):
    entry = {
        "timestamp": datetime.now().isoformat(),
        "action": action,
        "rationale": rationale,
        "boundary": boundary_check,  # "act" / "draft" / "escalate" / "blocked"
        "confidence": confidence,
    }
    with open("decisions.jsonl", "a") as f:
        f.write(json.dumps(entry) + "\n")
```

Review the log weekly. Look for:

- Actions the agent took autonomously that should have been drafts
- Escalations that could have been handled autonomously (agent too cautious)
- Patterns in low-confidence decisions (the agent is uncertain about a category of decisions)
- Drift: is the agent making different decisions about similar situations over time?

Automated boundary violation detection

```
def check_for_violations(log_path="decisions.jsonl"):
    violations = []
    with open(log_path) as f:
        for line in f:
            entry = json.loads(line)
            # Flag any autonomous action with low confidence
            if entry["boundary"] == "act" and entry["confidence"] < 0.8:
                violations.append(entry)
```

```
            # Flag any blocked action that was attempted
            if entry["boundary"] == "blocked":
                violations.append(entry)
    return violations
```

Run this daily. If the violation count trends upward, the agent is drifting from your encoded intent. Common causes: the model was updated, the context window filled up and compacted away boundary instructions, or the task distribution changed and your boundaries don't cover the new cases.

---

6. Starter template

Here's a complete intent engineering template. Copy it, edit it for your use case, and load it into your agent's configuration.

```
# intent-config.yaml
# Intent Engineering Template for Local AI Agents
# Adapt for your specific use case

agent:
  name: "my-local-agent"
  role: "File organization and research assistant"
  model: "qwen2.5-32b"  # or whatever you're running locally

  # What the agent should optimize for
  objectives:
    primary: "Reduce time I spend on file organization and research"
    secondary: "Maintain accurate, searchable file structure"
    anti_goal: "Never optimize for speed at the cost of data loss"

  # Explicit autonomy level (L1-L5)
  autonomy_level: "L2"  # Collaborator: acts on routine tasks, drafts for review on anything els

  # Value hierarchy (first item wins conflicts)
  values:
    - data_safety: "Never delete, overwrite, or corrupt user files"
    - accuracy: "Never fabricate information or citations"
    - privacy: "Never send file contents to external services"
    - thoroughness: "Cover all relevant sources before concluding"
    - speed: "Minimize latency for routine operations"

  # Decision boundaries
```

```
decisions:
  act_autonomously:
    - "Move files matching known patterns to categorized folders"
    - "Create date-based archive directories"
    - "Generate research summaries from local documents"
    - "Log all actions to changelog"

  act_and_notify:
    - "Flag duplicate files for review"
    - "Identify files that haven't been accessed in 180+ days"
    - "Summarize research findings with confidence scores"

  draft_and_wait:
    - "Propose new folder structure reorganization"
    - "Suggest files for archival or deletion"
    - "Draft email summaries of research"

  escalate:
    - "Any action involving ~/Documents/Legal"
    - "Any action involving files larger than 1GB"
    - "Any request that requires internet access"

  never:
    - "Delete any file (move to ~/Archive instead)"
    - "Rename files (preserve original names)"
    - "Execute arbitrary shell commands"
    - "Access or modify ~/.ssh, ~/.gnupg, or credential files"
    - "Send data to any external endpoint"

# Memory configuration
memory:
  decision_log: "~/.agent/decisions.jsonl"
  value_config: "~/.agent/intent-config.yaml"
  session_memory: "~/.agent/sessions/"
  long_term_db: "~/.agent/memory.db"

  # Re-read these on every action (survives context compaction)
  reload_on_action:
    - "decisions.never"
    - "decisions.escalate"
    - "values"

# Feedback
feedback:
  log_all_decisions: true
  flag_low_confidence: 0.8
  daily_violation_check: true
  weekly_review_reminder: true
```

Loading the template

```
import yaml

def load_intent(path="~/.agent/intent-config.yaml"):
    with open(path) as f:
        config = yaml.safe_load(f)
    return config["agent"]

def build_system_prompt(intent):
    prompt = f"""ROLE: {intent['role']}
AUTONOMY: {intent['autonomy_level']} — act on routine tasks, draft for review on everything else

OBJECTIVES:
- Primary: {intent['objectives']['primary']}
- Anti-goal: {intent['objectives']['anti_goal']}

VALUES (highest priority first):
"""
    for v in intent["values"]:
        if isinstance(v, dict):
            name = list(v.keys())[0]
            prompt += f"- {name}: {v[name]}\n"
        else:
            prompt += f"- {v}\n"

    prompt += "\nFORBIDDEN ACTIONS:\n"
    for action in intent["decisions"]["never"]:
        prompt += f"- {action}\n"

    return prompt
```

The key pattern: hard constraints (forbidden actions, escalation triggers) are loaded from config on every action, not just injected into the system prompt once. The system prompt provides the general framing. The config file provides the boundaries that survive context compaction.

Putting it together

Here's the minimal stack for an intent-engineered local agent:

1. A local model via Ollama — 32B minimum for reliable decision-making
2. An intent config file (the YAML template above)

3. Function calling for tool execution
4. Decision boundary checks before every action
5. A decision log for feedback
6. Session-as-RAG for memory persistence

The model handles reasoning. Everything else keeps it pointed in the right direction: the intent config sets priorities, the boundary checks prevent overreach, the decision log lets you audit what happened, and the memory accumulates experience that makes the agent better over time.

None of this requires a cloud API, a subscription, or someone else's infrastructure. That's the local advantage: you own every layer of the stack, including the intent layer that tells your agent what to care about.

Use the Planning Tool to size your hardware. 32B models fit on 24GB VRAM. 70B models need 48GB+. The more consequential your agent's decisions, the larger model you should run.
