n8n Agent Project — Pure JSON Contract (Option B)

A ready-to-ship project spec for an agent loop that works across any LLM by forcing **one strict JSON object** per turn. Drop this into n8n (or any orchestrator). Minimal moving parts, maximum determinism.

---

1) Goal & Scope

- **Goal:** Build a small, reliable agent that answers SMS/dashboard queries by calling a few safe tools.
- **Model-agnostic:** Works with Anthropic, OpenAI, Bedrock, etc.
- **Contract:** Model must output a **single JSON object** matching our schema. No prose.
- **Orchestrator:** n8n manages loops, state, budgets, and tool execution.

---

2) High-Level Architecture

```
[Trigger (Webhook/Twilio)]
   → [Load/Init State]
   → [LLM Call (Pure JSON prompt)]
   → [Validate Output]
      ├─ type=tool    → [Run Tool Node(s)] → [Append result to scratchpad]
      ├─ type=clarify → [Ask user a single question] → [Stop or await]
      └─ type=respond → [Return answer]
   → [Update Counters / Stop Conditions]
   → [Loop or Exit]
```

State fields (example):

- `goal, constraints`
- `plan_summary, progress_summary, latest_observations`
- counters: `steps_used, tool_calls_used, tokens_used`
- budgets: `max_steps, max_tool_calls, max_seconds`

Persist in n8n static data or external store (Redis/Postgres). For demos, memory can be ephemeral.

---

3) JSON Contract (Schema)

System Prompt (paste into your LLM call):

```
You are a bounded agent controller. Always output a SINGLE JSON object matching
this schema:
{
  "control": {
    "done": boolean,
    "reason": "ok" | "cannot_proceed" | "need_clarification"
  },
  "next_action": {
    "type": "tool" | "respond" | "clarify",
    "name": string,           // required if type=tool
    "args": object,           // required if type=tool (MUST match tool schema)
    "message": string         // required if type in {respond, clarify}
  },
  "state_update": {
    "plan": string,           // short plan update (<= 6 lines)
    "observation": string,    // what we just learned this turn
    "confidence": number      // 0.0..1.0
  }
}
Rules:
- Output ONLY that one JSON object. No markdown, no prose, no prefix/suffix.
- If a tool arg would fail validation, set next_action.type='respond' with
message detailing the bad field.
- Budgets (soft promises):
  - max_steps={{state.max_steps}}
  - max_tool_calls={{state.max_tool_calls}}
  - hard_stop_after_seconds={{state.max_seconds}}
- When blocked by missing inputs, use next_action.type='clarify' with ONE
concise question.
- If truly cannot proceed, set control.done=true and reason='cannot_proceed'
with a brief explanation.
```

User Prompt Template (first turn):

```
USER_REQUEST:
{{input.user_request}}

SCRATCHPAD (compact):
GOAL: {{state.goal}}
CONSTRAINTS: {{state.constraints}}
PLAN: {{state.plan_summary}}
PROGRESS: {{state.progress_summary}}
OBS: {{state.latest_observations}}
BUDGET_USED: steps={{state.steps_used}} tool_calls={{state.tool_calls_used}}
tokens={{state.tokens_used}}
```

```
TOOLS (strict JSON Schemas):
1) get_counts
schema = {
  "type":"object",
  "properties":{
    "start_date":{"type":"string","pattern":"^\\d{4}-\\d{2}-\\d{2}$"},
    "end_date":{"type":"string","pattern":"^\\d{4}-\\d{2}-\\d{2}$"},
    "label":{"type":"string","enum":["angry","praise","info"]}
  },
  "required":["start_date","end_date","label"],
  "additionalProperties":false
}

2) today_range
schema = { "type":"object", "properties":{}, "additionalProperties":false }

RULES:
- Choose EXACTLY ONE next_action per turn.
- If you need today’s dates, call today_range first.
- If blocked by missing inputs, use next_action.type='clarify' with a single
concise question.
- If truly cannot proceed, set control.done=true and reason='cannot_proceed' and
explain.
```

Example Model Output (expected):

```
{
  "control": {"done": false, "reason": "ok"},
  "next_action": {"type": "tool", "name": "today_range", "args": {}},
  "state_update": {
    "plan": "Confirm today window, then query counts for 'angry'.",
    "observation": "We lack date boundaries.",
    "confidence": 0.84
  }
}
```

---

4) n8n Wiring (Node Outline)

1. **Trigger**: Webhook (or Twilio → Webhook).
2. **Load/Init State**: Function node: fetch from DB or `getWorkflowStaticData('global')` / `'node'`.
3. **LLM Call**: HTTP Request node (POST to your provider) with System + User templates above.
4. **Validate Output**: Function node runs the validator below; throws on contract violations.
5. **Router** (IF/Switch on `next_action.type`):

6. `tool` → Tool nodes (HTTP/DB); capture result JSON → append to SCRATCHPAD `OBS`.
7. `clarify` → Respond a single question to user; **stop/await**.
8. `respond` → Return final answer; **exit**.
9. **Update State**: Function node increments counters; stores last tool call, time.
10. **Stop/Loop**: IF on hard caps; else go back to **LLM Call**.
11. **(Optional)** Compress Scratchpad every 2–3 steps (see §7).

---

5) Validator (Function Node, JS)

```
// Input: model_output in $json.model_output (string or object)
const raw = typeof $json.model_output === 'string' ? $json.model_output :
JSON.stringify($json.model_output);
let out;
try { out = typeof $json.model_output === 'string' ?
JSON.parse($json.model_output) : $json.model_output; }
catch (e) { throw new Error('LLM contract violation: not valid JSON'); }

function bad(msg){ throw new Error(`LLM contract violation: ${msg}`); }
if (!out || typeof out !== 'object') bad('not an object');
const { control, next_action, state_update } = out;
if (!control || typeof control.done !== 'boolean') bad('control.done missing');
if (!next_action || !['tool','respond','clarify'].includes(next_action.type))
bad('next_action.type invalid');
if (next_action.type === 'tool') {
  if (!next_action.name || typeof next_action.args !== 'object') bad('tool name/
args missing');
  // Allow-list
  const tools = { get_counts: true, today_range: true };
  if (!tools[next_action.name]) bad(`unknown tool: ${next_action.name}`);
}
if ((next_action.type === 'respond' || next_action.type === 'clarify') && !
next_action.message) {
  bad('message required for respond/clarify');
}
return [{ out, valid: true }];
```

---

6) Tool Nodes (Examples)

6.1 today_range (Function)

```
const today = new Date();
const yyyy = today.getFullYear();
const mm = String(today.getMonth()+1).padStart(2,'0');
const dd = String(today.getDate()).padStart(2,'0');
return [{ start_date: `${yyyy}-${mm}-${dd}`, end_date: `${yyyy}-${mm}-${dd}` }];
```

6.2 get_counts (HTTP Request → your API)

- **Method:** GET `https://api.example.local/counts?start={{$json.start_date}}&end={{$json.end_date}}&label={{$json.label}}`
- Capture JSON → store as `tool_result`.

Feed back to model (next turn): append to SCRATCHPAD `OBS`: `Counts: { label, value, start, end }`.

---

7) Scratchpad Compression Prompt (every 2–3 steps)

```
You are my compressor. Convert the RAW LOG below into a compact scratchpad (≤
600 tokens) with headers:
GOAL:
CONSTRAINTS:
PLAN: (3–6 numbered lines)
PROGRESS:
OBSERVATIONS: (bulleted, facts only)
OPEN_QUESTIONS:
RISKS:

RAW LOG:
{{state.raw_log_chunk}}
```

Use outputs to refresh `plan_summary`, `progress_summary`, `latest_observations`.

---

8) Budgets & Stop Conditions

- **Hard caps (n8n enforces):**
- `steps_used >= max_steps`
- `tool_calls_used >= max_tool_calls`
- `elapsed_seconds >= max_seconds`
- **Thrash detector:** same tool + identical args twice in a row → do ONE reflect turn; if still stuck, stop.
- **Clean exits:**
- `respond` action → final answer to user.
- `clarify` action → ask question and pause.
- `cannot_proceed` → explain and exit.

---

9) Env & Secrets

- `LLM_API_URL`, `LLM_API_KEY`
- Tool credentials (scoped, least privilege): `DASHBOARD_API_KEY`, etc.
- Timezone (for today_range): `TZ=America/New_York` (or handle upstream).

---

10) Testing & Telemetry

Golden tasks (at least 10):

- today's angry count; week range; missing label → clarify; invalid date → respond with error; etc.

Metrics to log per turn:

- step_index, next_action (type/name), tool_latency_ms, tokens_in/out, total_cost_est, confidence, done flag, reason.

Acceptance:

- ≥95% valid JSON
- ≤1 clarify per successful task on average
- ≤5 steps per solved task

10.1) Sequence-Aware Observability (Optional but Recommended)

Purpose. Make each agent turn traceable, replayable, and safe to retry across providers (including weaker/ HuggingFace models). Keep the ledger **outside** the model; feed only compact summaries back into prompts.

Core fields to capture per tool call / step

- `run_id` (ULID/UUID for the whole execution)
- `turn` (int), `plan_rev` (int)
- `action_id`, `parent_action_id` (ULIDs)
- `tool_call_seq` (monotonic int), `tool_name`
- `tool_args_hash` (sha256 of sorted args), `idempotency_key = tool_name|stable(args)`
- `retry_index` (0..n)
- `ts_start`, `ts_end`, `duration_ms`
- `outcome` (ok|error|timeout), `error_code`
- `budget_snapshot` ({steps_used, tool_calls_used, tokens_used})

  Do not let the LLM assign ids or sequence numbers. n8n (or your DB) is the source of truth.

Signals to compute from the ledger

- **Thrash:** same `tool_name + idempotency_key` in adjacent steps → do 1× reflect; if unchanged, stop or escalate once.
- **Stall:** 2 consecutive `need_clarification` or unusual `duration_ms` spike → ask user or exit.
- **Flaky tool:** rolling error_rate per tool > threshold → trip a circuit breaker.

Headers you can forward to tools (nice to have)

`X-Run-Id`, `X-Step-Id`, `X-Parent-Step`, `X-Idempotency-Key`.

Function node: Idempotency/Dedupe gate (pre-tool)

```
// Inputs expected: tool_name (string), tool_args (object), ttlSeconds (number,
optional)
function stable(obj){
  if (obj === null || typeof obj !== 'object') return obj;
  if (Array.isArray(obj)) return obj.map(stable);
  return Object.keys(obj).sort().reduce((o,k)=>{o[k]=stable(obj[k]);return o;},
{});
}
const name = $json.tool_name;
const args = stable($json.tool_args || {});
const key = `${name}|${JSON.stringify(args)}`;
const ttlMs = (($json.ttlSeconds ?? 300) * 1000);
const sd = getWorkflowStaticData('global');
sd._dedupe = sd._dedupe || {};
const now = Date.now();
// prune expired
for (const k of Object.keys(sd._dedupe)) if (sd._dedupe[k].expires_at < now)
delete sd._dedupe[k];
const hit = sd._dedupe[key];
if (hit && hit.expires_at >= now && hit.result) {
  // duplicate; short-circuit with cached result
  return [{ deduped: true, idempotency_key: key, cached_result: hit.result }];
}
// mark in-flight; the Event Logger will attach the result later
sd._dedupe[key] = { created_at: now, expires_at: now + ttlMs };
return [{ deduped: false, idempotency_key: key }];
```

Function node: Event logger (wrap every tool call)

```
/* Expects these fields on input $json:
run_id, turn, plan_rev, action_id, parent_action_id, tool_name,
idempotency_key, retry_index, ts_start, ts_end, duration_ms,
outcome, error_code, budget_snapshot, tool_result (object)
```

```
*/
function rid(){ return (Date.now().toString(36)
+Math.random().toString(36).slice(2,10)).toUpperCase(); }
const sd = getWorkflowStaticData('global');
sd._runs = sd._runs || {};
const run = sd._runs[$json.run_id] || { steps: [], created_at: new
Date().toISOString() };
const rec = {
  run_id: $json.run_id,
  turn: $json.turn ?? run.steps.length+1,
  plan_rev: $json.plan_rev ?? 0,
  action_id: $json.action_id || rid(),
  parent_action_id: $json.parent_action_id || null,
  tool_name: $json.tool_name,
  idempotency_key: $json.idempotency_key,
  retry_index: $json.retry_index ?? 0,
  tool_call_seq: run.steps.length + 1,
  ts_start: $json.ts_start || new Date().toISOString(),
  ts_end: $json.ts_end || new Date().toISOString(),
  duration_ms: $json.duration_ms ?? null,
  outcome: $json.outcome || 'ok',
  error_code: $json.error_code || null,
  budget_snapshot: $json.budget_snapshot || {}
};
run.steps.push(rec);
// Optional: cache successful results for idempotency short-circuiting
if (rec.outcome === 'ok' && $json.idempotency_key && $json.tool_result) {
  sd._dedupe = sd._dedupe || {};
  const entry = sd._dedupe[$json.idempotency_key];
  if (entry) entry.result = $json.tool_result;
}
sd._runs[$json.run_id] = run;
return [rec];
```

Prompt hygiene (don’t bloat tokens)

- Never paste the full ledger into the prompt. Keep a **compact scratchpad** (goal, constraints, plan, last observations).
- Summarize every 2–3 steps; link the ledger by `run_id` if you need deeper post-mortems.

Hardening for cheaper/weaker models (HuggingFace etc.)

- **Tight contract:** keep the Option-B schema; temperature 0–0.2; max_tokens small.
- **Strict parsing:** validate JSON; if parse fails, attempt a single JSON-repair pass; else treat as contract violation.
- **Output cap:** ask for short plans and one action per turn to avoid rambling.
- **Fallback:** on repeated violations, switch to a stronger model for 1 rescue turn, then drop back.

• **Tool allow-list & arg validation** in n8n regardless of provider.

10.2) Central Policy Node (Phase-1 Friendly Switchboard)

Purpose. Centralize model/provider settings so every LLM call and prompt read consistent caps & knobs. Lets you start simple and scale to multiple providers (Anthropic/OpenAI/HuggingFace) without refactors.

Policy Function node (drop-in)

```
// Global, editable policy; persists per-workflow via static data
const sd = getWorkflowStaticData('global');
sd.policy = sd.policy || {
  defaults: { temp_action: 0.1, temp_reply: 0.2, max_tokens: 600, max_steps:
5 },
  providers: {
    anthropic: { model: 'claude-3-5-sonnet-20241022', max_tokens: 800 },
    openai:    { model: 'gpt-4o-mini',                 max_tokens: 800 },
    hf:        { model: 'mistralai/Mixtral-8x7B-Instruct', max_tokens: 512 }
  },
  task_classes: {
    action: { temp: 0.1, max_tokens: 400 },
    reply:  { temp: 0.2, max_tokens: 800 }
  }
};

const provider = $json.provider || 'anthropic';
const task     = $json.task_class || 'action';

const base  = sd.policy.defaults;
const plat  = sd.policy.providers[provider] || {};
const taskp = sd.policy.task_classes[task] || {};

return [{
  policy: {
    model: plat.model,
    max_tokens: taskp.max_tokens ?? plat.max_tokens ?? base.max_tokens,
    temperature: taskp.temp ?? base.temp_action,
    max_steps: base.max_steps,
    max_tool_calls: 5,
    max_seconds: 30
  }
}];
```

Bindings

- **HTTP Request → Body params**: `model={{$json.policy.model}}`, `max_tokens={{$json.policy.max_tokens}}`, `temperature={{$json.policy.temperature}}`.
- **System prompt (soft budgets)**: `max_steps={{$json.policy.max_steps}}`, `max_tool_calls={{$json.policy.max_tool_calls}}`, `hard_stop_after_seconds={{$json.policy.max_seconds}}`.
- **Stop IF (hard budgets)**: `steps_used >= {{$json.policy.max_steps}}`, `tool_calls_used >= {{$json.policy.max_tool_calls}}`.

  Keep the Option-B JSON **contract unchanged**. Only the dial values come from `policy`.

---

11) Example cURL (smoke test)

```
curl -s -X POST "$LLM_API_URL"
 -H "Authorization: Bearer $LLM_API_KEY"
 -H "Content-Type: application/json"
 -d '{
  "model": "<your-model>",
  "max_tokens": 600,
  "temperature": 0,
  "system": "<paste system prompt>",
  "messages": [{"role":"user","content":"<paste user template with state>"}]
 }'
```

---

12) Security & Guardrails

- Enforce tool allow-list and schema validation in n8n before any side-effecting call.
- Read-only tools for demos; human-in-the-loop for writes.
- Rate-limit inbound triggers; add auth/secret for webhooks.

---

13) Optional: One-Shot Escalation

- Detect thrash or contradictions → swap the next *single* turn to a stronger model (e.g., Opus) using the same contract → drop back.

---

14) Checklist (Copy/Paste)

- [ ] Trigger node (Webhook/Twilio) with auth

- [ ] State init Function node (create `run_id`, counters, scratchpad)
- [ ] **Policy Function node** (central caps & knobs)
- [ ] LLM HTTP node (Option-B System/User prompts) reading from `policy`
- [ ] Validator Function node (contract checks)
- [ ] Idempotency/Dedupe gate Function (from §10.1)
- [ ] Tool nodes (today_range, get_counts) wrapped with Event logger Function
- [ ] Router IF node (`tool` | `respond` | `clarify`)
- [ ] Stop/Loop IF node (budgets + thrash detector)
- [ ] Optional Compressor LLM node (scratchpad maintenance)
- [ ] Metrics export (HTTP/DB) using ledger records

---

15) Next Steps (Tailoring)

- Swap `get_counts` to your real dashboard API.
- Add more tools (allow-list + schema): `get_top5`, `get_summary`, `get_by_segment`.
- Wire Twilio bidirectional SMS (clarify prompts).
- Add a small HTML/Slack responder node for non-SMS channels.