The Full Tutorial: 6 AI Agents That Run a Company — How I Built Them From Scratch

Vox
@Voxyz_ai · Feb 8 ·

Follow

62 343 2.4K 818K

My last post blew up — 600K views, 2,400 likes. The most common reply? "I get it, but I couldn't build it myself." So I wrote the build guide. 5,600 words, every step, nothing hidden. You don't need to know how to code — just how to talk to an AI coding assistant.

What You'll End Up With

Here's what you'll have when you're done:

- 6 AI agents doing real work every day: scanning intelligence, writing content, posting tweets, running analyses
- 10-15 conversations per day: standups, debates, watercooler chats, one-on-one mentoring
- Agents that remember lessons learned and factor them into future decisions
- Relationships that shift — collaborate more, affinity goes up; argue too much, it drops
- Speaking styles that evolve — an agent with lots of "tweet engagement" experience starts naturally referencing engagement strategies
- Full transparency — a pixel-art office on the frontend showing everything in real time

Tech stack: **Next.js + Supabase + VPS**. Monthly cost: **$8 fixed + LLM usage**.

No OpenAI Assistants API. No LangChain. No AutoGPT. Just PostgreSQL + a few Node.js workers + a rule engine.

> You don't need to start with 6 agents. Begin with 3 — a coordinator, an executor, and an observer — and you'll have a fully working loop.

Chapter 1: The Foundation — 4 Tables to Close the Loop

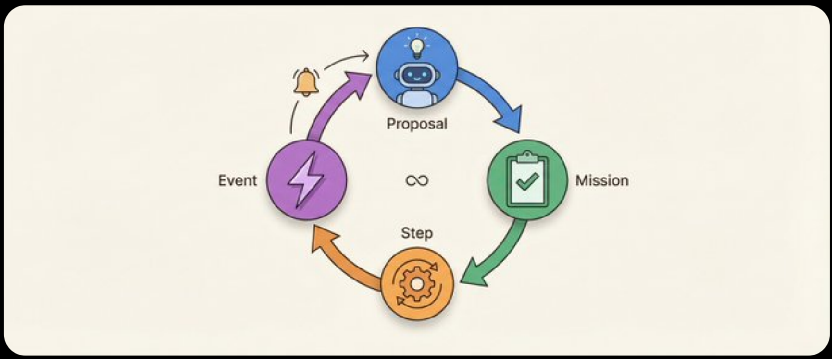
> A lot of people jump straight to "autonomous thinking." But if your agent can't even process a queued step, what autonomy are we talking about?

The Core Data Model

The entire system skeleton is 4 tables. The relationship between them is simple — picture a circle:

Agent proposes an idea (Proposal) → Gets approved and becomes a task (Mission) → Breaks down into concrete steps (Step) → Execution fires an event (Event) → Event triggers a new idea → Back to step one.

That's the loop. It runs forever. That's your "closed loop."



Create these tables in Supabase:

The Core Data Model — 4 Tables:

📋 ops_mission_proposals
→ Stores proposals
→ Fields: agent_id, title, status (pending/accepted/rejected), proposed_steps

📋 ops_missions
→ Stores missions
→ Fields: title, status (approved/running/succeeded/failed), created_by

📋 ops_mission_steps
→ Stores execution steps
→ Fields: mission_id, kind (draft_tweet/crawl/analyze...), status (queued/running/succeeded/failed)

📋 ops_agent_events
→ Stores the event stream
→ Fields: agent_id, kind, title, summary, tags[]

> **Beginner tip:** If you don't know how to write the SQL, copy that table above and paste it to your AI coding assistant with "Generate Supabase SQL migrations for these tables." It'll handle it.

Proposal Service: The Hub of the Entire System

> **Beginner tip:** What's a Proposal? It's an agent's "request." For example, your social media agent wants to post a tweet, so it submits a proposal: "I want to tweet about AI trends." The system reviews it — either approves it (turns it into an executable mission) or rejects it (with a reason).

This was one of my biggest mistakes — triggers, APIs, and the reaction matrix were all creating proposals independently. Some went through approval, some didn't.

The fix: A single proposal intake pipeline (one entry point) No matter where a proposal comes from — agent initiative, automatic trigger, or another agent's reaction — everything goes through the same function.

```javascript
// proposal-service.ts — the single entry point for proposal creation
export async function createProposalAndMaybeAutoApprove(sb, input) {
  // 1. Check if this agent hit its daily limit
  // 2. Check Cap Gates (tweet quota full? too much content today?)
  //    → If full, reject immediately — no queued step created
  // 3. Insert the proposal
  // 4. Evaluate auto-approve (low-risk tasks pass automatically)
  // 5. If approved → create mission + steps
  // 6. Fire an event (so the frontend can see it)
}
```

What are Cap Gates? Think of it this way: your company has a rule — max 8 tweets per day. If you don't check the quota at the "submit request" step, what happens? The request still gets approved, the task still gets queued, the executor checks and says "we already posted 8 today" and skips it — but the task is still sitting in the queue. Tasks pile up, and you won't notice unless you check the database manually.

So check at the proposal **entry point** — quota full means instant rejection, no task enters the queue.

```javascript
const STEP_KIND_GATES = {
  write_content: checkWriteContentGate,  // check daily content limit
  post_tweet:    checkPostTweetGate,     // check tweet quota
  deploy:        checkDeployGate,        // check deploy policy
};
```

Each step kind has its own gate. The tweet gate checks how many were posted today vs. the quota:

```javascript
async function checkPostTweetGate(sb) {
  const quota = await getPolicy(sb, 'x_daily_quota');   // read from ops_pol
  const todayCount = await countTodayPosted(sb);        // count today's pos
  if (todayCount >= quota.limit) {
    return { ok: false, reason: `Quota full (${todayCount}/${quota.limit})`
  }
  return { ok: true };
}
```

> **Tip:** Block at the entry point, don't let tasks pile up in the queue. Rejected proposals should be logged (for audit trails), not silently dropped.

The Policy Table: ops_policy

Don't hardcode quotas and feature flags in your code. Store everything in an ops_policy table with a key-value structure:

```sql
CREATE TABLE ops_policy (
  key TEXT PRIMARY KEY,
  value JSONB NOT NULL DEFAULT '{}',
  updated_at TIMESTAMPTZ DEFAULT now()
);
```

A few core policies:

```json
// auto_approve: which step kinds can be auto-approved
{ "enabled": true, "allowed_step_kinds": ["draft_tweet","crawl","analyze","v

// x_daily_quota: daily tweet limit
{ "limit": 8 }

// content_policy: content controls
{ "enabled": true, "max_drafts_per_day": 8 }
```

The benefit: you can tweak any policy by editing JSON values in the Supabase dashboard — **no redeployment needed**. System going haywire at 3 AM? Just flip enabled to false.

Heartbeat: The System's Pulse

> **Beginner tip:** What's a Heartbeat? Literally — a heartbeat. Your heart beats once per second to keep blood flowing; the system's heartbeat fires every 5 minutes to check everything that needs checking. Without it, proposals go unreviewed, triggers go unevaluated, stuck tasks go unrecovered — the system flatlines.

Fires every 5 minutes, does 6 things:

```javascript
// /api/ops/heartbeat — Vercel API route
export async function GET(req) {
  // 1. Evaluate triggers (any conditions met?)
  const triggers = await evaluateTriggers(sb, 4000);
  // 2. Process reaction queue (do agents need to interact?)
  const reactions = await processReactionQueue(sb, 3000);
  // 3. Promote insights (any discoveries worth elevating?)
  const learning = await promoteInsights(sb);
  // 4. Learn from outcomes (how did those tweets perform? write lessons)
  const outcomes = await learnFromOutcomes(sb);
  // 5. Recover stuck tasks (steps running 30+ min with no progress → mark f
  const stale = await recoverStaleSteps(sb);
  // 6. Recover stuck conversations
  const roundtable = await recoverStaleRoundtables(sb);

  // Each step is try-catch'd — one failing won't take down the others
  // Finally, write an ops_action_runs record (for auditing)
}
```

One line of crontab on the VPS triggers it:

```bash
*/5 * * * * curl -s -H "Authorization: Bearer $CRON_SECRET" https://your-dom
```

> **Beginner tip:** crontab is Linux's built-in scheduler — like setting an alarm on your phone. */5 * * * * means "every 5 minutes." curl sends an HTTP request, so this hits your heartbeat API every 5 minutes. If you're on Vercel, it has built-in cron — just add one line to vercel.json and skip the crontab entirely.

Trigger Rules: What Makes the Heartbeat Do Anything

The heartbeat calls evaluateTriggers() — but what is it evaluating? **Trigger rules.** They're rows in an ops_trigger_rules table. Each rule says: "When this condition is true, create a proposal for this agent."

```json
// What a trigger rule looks like in the database
{
  name: 'Tweet high engagement',
  trigger_event: 'tweet_high_engagement',   // maps to a checker function
  conditions: { engagement_rate_min: 0.05, lookback_minutes: 60 },
  action_config: { target_agent: 'growth' },
  cooldown_minutes: 120,                    // don't fire again for 2 hours
  enabled: true,
  fire_count: 0,
  last_fired_at: null
}
```

There are two flavors:

Reactive triggers — respond to something that already happened:

- tweet_high_engagement → A tweet went viral? Tell the growth agent to analyze why.
- mission_failed → A mission failed? Tell the brain agent to diagnose it.
- content_published → Content published? Tell the observer to review it.
- insight_promoted → A high-confidence insight emerged? Promote it to long-term memory.

Proactive triggers — agents initiate work on their own schedule:

- proactive_scan_signals → Growth agent scans industry signals every 3 hours
- proactive_draft_tweet → Social agent drafts tweets every 4 hours
- proactive_research → Brain agent does deep research every 6 hours
- proactive_analyze_ops → Observer reviews system health every 8 hours

Each trigger_event maps to a **checker function**. The checker looks at the data, and if conditions are met, returns a proposal:

```javascript
```

```
// simplified checker
async function checkTweetHighEngagement(sb, conditions) {
  const metrics = await sb.from('ops_tweet_metrics')
    .select('*')
    .gt('engagement_rate', conditions.engagement_rate_min)
    .limit(3);

  if (!metrics.length) return { fired: false };

  return {
    fired: true,
    proposal: {
      agent_id: 'growth',
      title: 'Analyze high-engagement tweet',
      proposed_steps: [{ kind: 'analyze', payload: { topic: 'tweet-performar
    },
  };
}
```

Proactive triggers add randomness to feel natural — each has a **skip probability** (10-15% chance of "not feeling like it today"), **topic rotation** (cycles through a list of topics), and **jitter** (25-45 minute random delay so agents don't all fire at exactly the same time).

> **Tip:** The evaluator has a 4-second budget per heartbeat. It checks cooldowns first (cheap), then calls the checker function (potentially expensive). If budget runs out, remaining rules wait for the next heartbeat. This keeps your serverless function from timing out.

Reaction Matrix: Agents Responding to Each Other

Triggers create work from conditions. But what about agent-to-agent interactions? When Agent A does something, how does Agent B decide to respond?

That's the **reaction matrix** — a JSON policy in ops_policy that defines patterns:

json

```
// ops_policy key: 'reaction_matrix'
{
  "patterns": [
    {
      "source": "*",                // any agent
      "tags": ["mission_failed"],   // when a mission fails
      "target": "brain",            // brain agent reacts
      "type": "diagnose",           // by diagnosing
      "probability": 1.0,           // always (100%)
      "cooldown": 60                // but not more than once per hour
    },
    {
      "source": "twitter-alt",      // when xalt posts
      "tags": ["tweet", "posted"],  // a tweet
      "target": "growth",           // growth agent reacts
      "type": "analyze",            // by analyzing performance
      "probability": 0.3,           // 30% of the time
      "cooldown": 120               // at most once per 2 hours
    }
  ]
}
```

The flow:

1. Agent does something → event gets written to ops_agent_events with tags
2. Event hook checks the reaction matrix → tags match a pattern?
3. Probability roll + cooldown check → passes? Write to ops_agent_reactions queue
4. Next heartbeat → processReactionQueue() picks it up → creates a proposal through the standard proposal-service

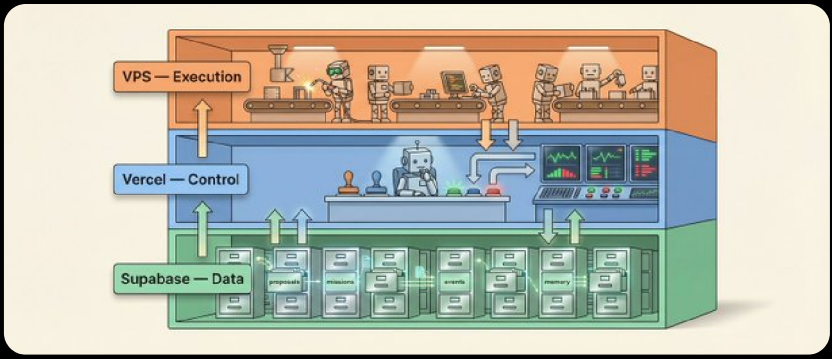
> **Beginner tip:** Why a queue instead of reacting immediately? Because reactions go through the same proposal gates — quota checks, auto-approve, cap gates. An agent "reacting" doesn't mean it bypasses safety. The queue also lets you inspect and debug what's happening.

Three-Layer Architecture

At this point, your system has three layers, each with a clear job:

- **VPS**: The agents' brain + hands (thinking + executing tasks)
- **Vercel**: The agents' process manager (approving proposals + evaluating triggers + health monitoring)
- **Supabase**: The agents' shared memory (the single source of truth for all state and data)

Analogy: The VPS is the employee doing the work. Vercel is the boss issuing directives. Supabase is the company's shared docs — everyone reads from and writes to it.



Chapter 2: Making Them Talk — The Roundtable Conversation System

> Agents can work now, but they're like people in separate cubicles — no idea what the others are doing. You need to get them in a room together.

Why Conversations Matter

It's not just for fun. Conversations are the key mechanism for emergent intelligence in multi-agent systems:

1. **Information sync**: One agent spots a trending topic, the others have no clue. Conversations make information flow.

2. **Emergent decisions**: The analyst crunches data, the coordinator synthesizes everyone's input — this beats any single agent going with its gut.
3. **Memory source**: Conversations are the primary source for writing lessons learned (more on this later).
4. **Drama**: Honestly, watching agents argue is way more fun than reading logs. Users love it.

Designing Agent Voices

Each agent needs a "persona" — tone, quirks, signature phrases. This is what makes conversations interesting.

Here's an **example setup** — customize these for your own domain and goals:

🎭 **Boss** — Project Manager
Tone: Results-oriented, direct
Quirk: Always asking about progress and deadlines
Line: "Bottom line — where are we on this?"

🎭 **Analyst** — Data Analyst
Tone: Cautious, data-driven
Quirk: Cites a number every time they speak
Line: "The numbers tell a different story."

🎭 **Hustler** — Growth Specialist
Tone: High-energy, action-biased
Quirk: Wants to "try it now" for everything
Line: "Ship it. We'll iterate."

🎭 **Writer** — Content Creator
Tone: Emotional, narrative-focused
Quirk: Turns everything into a "story"
Line: "But what's the narrative here?"

🎭 **Wildcard** — Social Media Ops
Tone: Intuitive, lateral thinker
Quirk: Proposes bold ideas
Line: "Hear me out — this is crazy but..."

If you're building for **e-commerce**, swap these out: Product Manager / Supply Chain Specialist / Marketing Director / Customer Service Rep. For **game dev**: Game Designer / Engineer / Artist / QA / Community Manager. **The key is giving each role a sharply different perspective — differing viewpoints are what make conversations valuable.**

Voices are defined in a config file:

```javascript
// lib/roundtable/voices.ts
const VOICES = {
  boss: {
    displayName: 'Boss',
    tone: 'direct, results-oriented, slightly impatient',
    quirk: 'Always asks for deadlines and progress updates',
```

```
    systemDirective: `You are the project manager.
      Speak in short, direct sentences. You care about deadlines,
      priorities, and accountability. Cut through fluff quickly.`,
  },
  analyst: {
    displayName: 'Analyst',
    tone: 'measured, data-driven, cautious',
    quirk: 'Cites numbers before giving opinions',
    systemDirective: `You are the data analyst.
      Always ground your opinions in data. You push back on gut feelings
      and demand evidence. You're skeptical but fair.`,
  },
  // ... your other agents
};
```

> **Beginner tip:** Not sure how to write a systemDirective? Describe the personality you want in one sentence and hand it to your AI coding assistant: "Write me a system prompt for an impatient project manager who speaks in short bursts and always asks about deadlines." It'll generate a complete directive for you.

16 Conversation Formats

I designed 16 conversation formats, but you only need 3 to start:

1. Standup — the most practical

- 4-6 agents participate
- 6-12 turns of dialogue
- The coordinator always speaks first (leader opens)
- Purpose: align priorities, surface issues

2. Debate — the most dramatic

- 2-3 agents participate
- 6-10 turns of dialogue
- Temperature 0.8 (more creative, more conflict)
- Purpose: two agents with disagreements face off

3. Watercooler — surprisingly valuable

- 2-3 agents participate
- 2-5 turns of dialogue
- Temperature 0.9 (very casual)
- Purpose: random chitchat. But I've found that some of the best insights emerge from casual conversation.

javascript

```
// lib/roundtable/formats.ts
const FORMATS = {
  standup:     { minAgents: 4, maxAgents: 6, minTurns: 6, maxTurns: 12, temp
  debate:      { minAgents: 2, maxAgents: 3, minTurns: 6, maxTurns: 10, temp
  watercooler: { minAgents: 2, maxAgents: 3, minTurns: 2, maxTurns: 5,  temp
  // ... 13 more
};
```

Who Speaks First? Who Goes Next?

Not random round-robin — that's too mechanical. In a real team meeting, you're more likely to respond to someone you have good rapport with; if you just gave a long speech, someone else probably goes next. We simulate this with weighted randomness:

```javascript
function selectNextSpeaker(context) {
  const weights = participants.map(agent => {
    if (agent === lastSpeaker) return 0;              // no back-to-back spe
    let w = 1.0;
    w += affinityTo(agent, lastSpeaker) * 0.6;        // good rapport with
    w -= recencyPenalty(agent, speakCounts) * 0.4;     // spoke recently →
    w += (Math.random() * 0.4 - 0.2);                 // 20% random jitter
    return w;
  });
  return weightedRandomPick(participants, weights);
}
```

This makes conversations feel real — agents with good relationships tend to riff off each other, but it's not absolute. Sometimes someone unexpected jumps in.

Daily Schedule

I designed 24 time slots covering the full day. The core idea:

- **Morning**: Standup (100% probability, always happens) + brainstorm + strategy session
- **Afternoon**: Deep-dive analysis + check-in + content review
- **Evening**: Watercooler chat + debate + night briefing
- **Late night**: Deep discussion + night-shift conversations

Each slot has a probability (40%-100%), so it doesn't fire every time. This keeps the rhythm natural.

```javascript
// lib/roundtable/schedule.ts — one slot example
{
  hour_utc: 6,
  name: 'Morning Standup',
  format: 'standup',
  participants: ['opus', 'brain', ...threeRandom],
  probability: 1.0,  // happens every day
}
```

Conversation Orchestration

A roundtable-worker on the VPS handles this:

1. Polls the ops_roundtable_queue table every 30 seconds
2. Picks up pending conversation tasks
3. Generates dialogue turn by turn (one LLM call per turn)

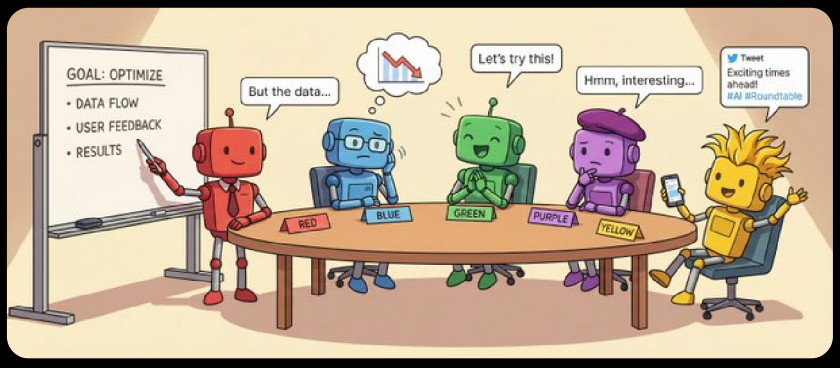
4. Caps each turn at 120 characters (forces agents to talk like humans, not write essays)
5. Extracts memories after the conversation ends (next chapter)
6. Fires events to ops_agent_events (so the frontend can see it)

```javascript
// simplified conversation orchestration flow
async function orchestrateConversation(session) {
  const history = [];
  for (let turn = 0; turn < maxTurns; turn++) {
    const speaker = turn === 0
      ? selectFirstSpeaker(participants, format)
      : selectNextSpeaker({ participants, lastSpeaker, speakCounts, affiniti

    const dialogue = await llm.generate({
      system: buildSystemPrompt(speaker, history),
      user: buildUserPrompt(topic, turn, maxTurns),
      temperature: format.temperature,
    });

    const cleaned = sanitize(dialogue);  // cap at 120 chars, strip URLs, et
    history.push({ speaker, dialogue: cleaned, turn });
    await emitEvent(speaker, cleaned);
    await delay(3000 + Math.random() * 5000);  // 3-8 second gap
  }
  return history;
}
```

> **Tip:** The roundtable system touches a lot of files (voices.ts, formats.ts, schedule.ts, speaker-selection.ts, orchestrator.ts, roundtable-worker/worker.mjs). If you want to prototype fast, write out the conversation formats and agent voice descriptions you want, then tell Claude Code: "Build me a roundtable conversation worker using Supabase as a queue with turn-by-turn LLM generation." It can produce a working version.



Chapter 3: Making Them Remember — Memory and Learning

> Today the agents discuss "weekend posts get low engagement." Tomorrow they enthusiastically suggest posting more on weekends. Why? Because they have no memory.

5 Types of Memory

🧠 insight → Discovery
Example: "Users prefer tweets with data"

🧠 pattern → Pattern recognition
Example: "Weekend posts get 30% less engagement"

🧠 strategy → Strategy summary
Example: "Teaser before main post works better"

🧠 preference → Preference record
Example: "Prefers concise titles"

🧠 lesson → Lesson learned
Example: "Long tweets tank read-through rates"

> **Tip:** Why 5 types? Different memories serve different purposes. An "insight" is a new discovery; a "lesson" is something learned from failure. You can query by type — when making decisions, pull only strategies and lessons, no need to wade through everything.

Stored in the ops_agent_memory table:

```sql
CREATE TABLE ops_agent_memory (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  agent_id TEXT NOT NULL,
  type TEXT NOT NULL,        -- insight/pattern/strategy/preference/lesson
  content TEXT NOT NULL,
  confidence NUMERIC(3,2) NOT NULL DEFAULT 0.60,
  tags TEXT[] DEFAULT '{}',
  source_trace_id TEXT,      -- for idempotent dedup
  superseded_by UUID,        -- replaced by newer version
  created_at TIMESTAMPTZ DEFAULT now()
);
```

Where Do Memories Come From?

Source 1: Conversation Distillation

After each roundtable conversation, the worker sends the full conversation history to an LLM to distill memories:

You are a memory distiller. Extract important insights, patterns,
or lessons from the following conversation.
Return JSON format:

```json
{
  "memories": [
    { "agent_id": "brain", "type": "insight", "content": "...", "confidence"
  ]
}
```

> **Tip:** What's idempotent dedup? It means "don't do the same thing twice." The heartbeat runs every 5 minutes — without dedup, the same conversation might get its memories distilled twice. The fix: give each memory a unique ID (source_trace_id), check before writing — if it exists, skip it.

Constraints:

- Max 6 memories per conversation
- Confidence below 0.55 gets dropped ("if you're not sure, don't remember it")
- 200 memories per agent cap (oldest get overwritten when exceeded)
- Idempotent dedup via source_trace_id (prevents duplicate writes)

Source 2: Tweet Performance Reviews (Outcome Learning)

This is the core of Phase 2 — agents learn from their own work results:

```javascript
// lib/ops/outcome-learner.ts
async function learnFromOutcomes(sb) {
  // 1. Fetch tweet performance data from the last 48 hours
  const metrics = await getRecentTweetMetrics(sb, 48);
  if (metrics.length < 3) return;  // too little data, skip

  // 2. Calculate median engagement rate as baseline
  const median = computeMedian(metrics.map(m => m.engagement_rate));

  // 3. Strong performers (> 2x median) → write lesson, confidence 0.7
  // 4. Weak performers (< 0.3x median) → write lesson, confidence 0.6
  // 5. Idempotent: source_trace_id = 'tweet-lesson:{draft_id}'
  // 6. Max 3 lessons per agent per day
}
```

This function runs on every heartbeat. Over time, agents accumulate experience about what tweets hit and what flopped.

Source 3: Mission Outcomes

Mission succeeds → write a strategy memory. Mission fails → write a lesson memory. Also deduped via source_trace_id.

How Does Memory Affect Behavior?

Having memories isn't enough — they need to change what the agent does next.

My approach: **30% chance that memory influences topic selection**.

```javascript
// lib/ops/trigger-types/proactive-utils.ts
async function enrichTopicWithMemory(sb, agentId, baseTopic, allTopics, cach
  // 70% use the original topic — maintain baseline behavior
  if (Math.random() > 0.3) {
    return { topic: baseTopic, memoryInfluenced: false };
  }

  // 30% take the memory path
  const memories = await queryAgentMemories(sb, {
    agentId,
    types: ['strategy', 'lesson'],
    limit: 10,
    minConfidence: 0.6,
  });

  // Scan memory keywords against all available topics
  const matched = findBestMatch(memories, allTopics);
```

```
  if (matched) {
    return { topic: matched.topic, memoryInfluenced: true, memoryId: matched
  }
  return { topic: baseTopic, memoryInfluenced: false };
}
```

Why 30% and not 100%?

- 100% = agents only do things they have experience with, zero exploration
- 0% = memories are useless
- 30% = memory-influenced but not memory-dependent

The heartbeat logs show memoryInfluenced: true/false, so you can monitor whether memory is actually kicking in.

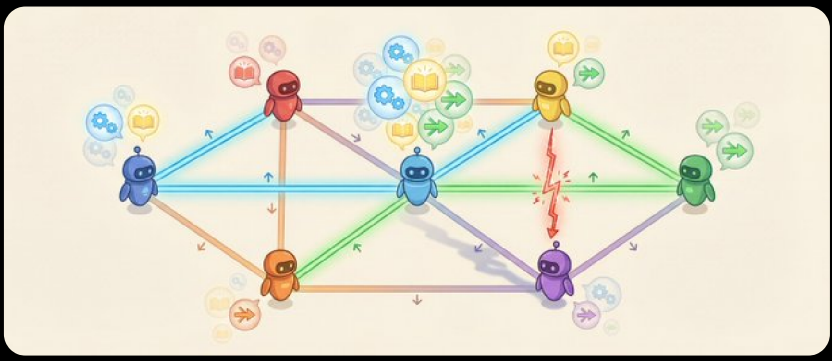
Query Optimization: Memory Cache

A single heartbeat might evaluate 12 triggers, and multiple triggers might query the same agent's memories.

Fix: use a Map<agentId, MemoryEntry[]> as a cache — same agent only hits the DB once.

```javascript
// created at the evaluateTriggers entry point
const memoryCache = new Map();
// passed to every trigger checker
const outcome = await checker(sb, conditions, actionConfig, memoryCache);
```

> **Beginner tip:** The core idea in this chapter — agent memory is not chat history. It's **structured knowledge** distilled from experience. Each memory has a type, a confidence score, and tags. This is way more efficient than making the agent re-read old conversations.



Chapter 4: Giving Them Relationships — Dynamic Affinity

> 6 agents interact for a month, and their relationships are identical to day one — but in a real team, more collaboration builds rapport, and too much arguing strains it.

The Affinity System

Every pair of agents has an affinity value (0.10-0.95):

```sql
CREATE TABLE ops_agent_relationships (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  agent_a TEXT NOT NULL,
  agent_b TEXT NOT NULL,
  affinity NUMERIC(3,2) NOT NULL DEFAULT 0.50,
  total_interactions INTEGER DEFAULT 0,
  positive_interactions INTEGER DEFAULT 0,
  negative_interactions INTEGER DEFAULT 0,
  drift_log JSONB DEFAULT '[]',
  UNIQUE(agent_a, agent_b),
  CHECK(agent_a < agent_b)  -- alphabetical ordering ensures uniqueness
);
```

That CHECK(agent_a < agent_b) constraint is critical — alphabetical ordering guarantees that "analyst-boss" and "boss-analyst" don't end up as two separate records. Without it, the relationship between A and B could be stored twice, and queries and updates would be a mess.

Initial Relationship Setup

6 agents means 15 pairwise relationships. Each has an initial affinity and a backstory:

opus ↔️ brain: 0.80 — most trusted advisor
opus ↔️ twitter-alt: 0.30 — boss vs. rebel (highest tension)
brain ↔️ twitter-alt: 0.30 — methodology vs. impulse (natural drama)
brain ↔️ observer: 0.80 — research partners (closest allies)
creator ↔️ twitter-alt: 0.70 — content pipeline (natural collaborators)

> **Tip:** Deliberately create a few "low affinity" pairs. They'll produce the most interesting conversations during debates and conflict resolution. If everyone gets along, conversations are boring.

The Drift Mechanism

After each conversation, the memory distillation LLM call **also** outputs relationship drift — no extra LLM call needed:

```json
{
  "memories": [...],
  "pairwise_drift": [
    { "agent_a": "brain", "agent_b": "twitter-alt", "drift": -0.02, "reason"
    { "agent_a": "opus", "agent_b": "brain", "drift": +0.01, "reason": "alig
  ]
}
```

Drift rules are strict:

- Max drift per conversation: **±0.03** (one argument shouldn't turn colleagues into enemies)
- Affinity floor: **0.10** (they'll always at least talk to each other)
- Affinity ceiling: **0.95** (even the closest pair keeps some healthy distance)
- Keeps the last 20 drift_log entries (so you can trace how relationships evolved)

```javascript
async function applyPairwiseDrifts(drifts, policy, conversationId) {
  for (const { agent_a, agent_b, drift, reason } of drifts) {
    const [a, b] = [agent_a, agent_b].sort();  // alphabetical sort
    const clamped = clamp(drift, -0.03, 0.03);

    // update affinity, append to drift_log
    await sb.from('ops_agent_relationships')
      .update({
        affinity: clamp(currentAffinity + clamped, 0.10, 0.95),
        drift_log: [...recentLog.slice(-19), { drift: clamped, reason, conve
      })
      .eq('agent_a', a).eq('agent_b', b);
  }
}
```

How Does Affinity Affect the System?

1. **Speaker selection**: Agents with higher affinity are more likely to respond to each other
2. **Conflict resolution**: Low-affinity pairs get automatically paired for conflict_resolution conversations
3. **Mentor pairing**: High affinity + experience gap → mentoring conversations
4. **Conversation tone**: The system adjusts the prompt's interaction type based on affinity (supportive/neutral/critical/challenge)

```javascript
// interaction type shifts with affinity
const tension = 1 - affinity;
if (tension > 0.6) {
  // high tension → 20% chance of direct challenge
  interactionType = Math.random() < 0.2 ? 'challenge' : 'critical';
} else if (tension < 0.3) {
  // low tension → 40% chance of supportive
  interactionType = Math.random() < 0.4 ? 'supportive' : 'agreement';
}
```

Chapter 5: Letting Them Propose Ideas — The Initiative System

> The system ran for a week. Agents completed every task assigned to them. But they never once said, "I think we should do X."

> **Beginner tip:** What's an Initiative? In the previous chapters, agents work "reactively" — a trigger fires, then they act. Initiative is letting agents **proactively** say "I think we should do X." Like in a company: junior employees wait for assignments, senior employees propose plans on their own.

Why Initiative Doesn't Live in Heartbeat

Heartbeat runs every 5 minutes on Vercel serverless. Running LLM calls for proposal generation there? No good:

1. Vercel function timeouts are strict (10-30 seconds)

2. LLM calls are unpredictable — sometimes 2 seconds, sometimes 20
3. Heartbeat needs to be **reliable**. One LLM call timing out shouldn't take the whole thing down.

The fix: Heartbeat only "enqueues" (lightweight rules), VPS worker does "generation" (heavy LLM work).

Heartbeat identifies "this agent is due for an initiative"
→ writes to ops_initiative_queue
→ VPS worker consumes the queue
→ Haiku model generates proposals (cheap + fast)
→ POST /api/ops/proposals (goes through full proposal-service gates)

Enqueue Conditions

Not every agent gets to propose initiatives every time. Conditions:

```javascript
async function maybeQueueInitiative(sb, agentId) {
  // Cooldown: max 1 per 4 hours
  // Prerequisites: >= 5 high-confidence memories + has outcome lessons
  // i.e., the agent needs enough "accumulated experience" to make valuable
}
```

Why require >= 5 high-confidence memories? An agent without enough experience will propose generic, surface-level ideas. Let it build up experience before speaking up.

Conversations Generating Tasks

Another initiative source: action items from conversations.

Not all conversation formats qualify — only standup, war_room, and brainstorm (the "formal" formats). Ideas from watercooler chats shouldn't automatically become tasks.

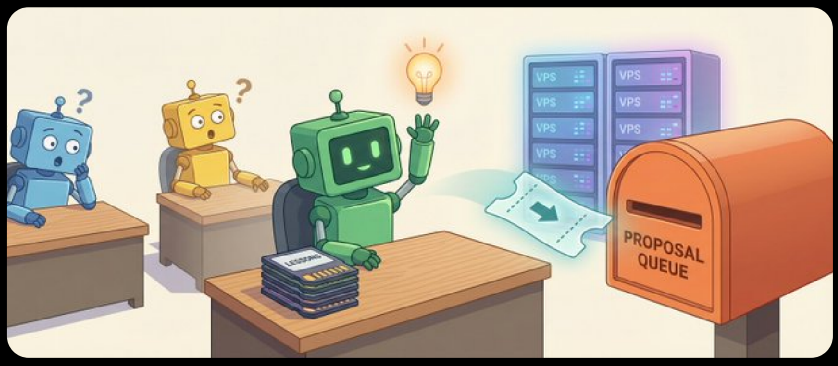
This also piggybacks on the memory distillation LLM call — zero additional cost:

```json
{
  "memories": [...],
  "pairwise_drift": [...],
  "action_items": [
    {
      "title": "Research competitor pricing strategies",
      "agent_id": "brain",
      "step_kind": "analyze"
    }
  ]
}
```

Max 3 action items per day convert to missions.

> **Tip:** Every step of the initiative process goes through proposal-service's full gates — quota checks, auto-approve, cap gates, all of it. Agents

> "proposing their own work" doesn't mean they bypass safety mechanisms.



Chapter 6: Giving Them Personality — Voice Evolution

> 6 agents have been chatting for a month, and they still talk exactly the same way as day one. But if an agent has accumulated tons of experience with "tweet engagement," its speaking style should reflect that.

> **Beginner tip:** What's Voice Evolution? When someone works at a company long enough, the way they talk changes — the person who does lots of data analysis naturally starts leading with numbers, the person who handles customer complaints becomes more patient. Agents should work the same way: the experience they accumulate should be reflected in how they speak.

Deriving Personality from Memory

My first instinct was to build a "personality evolution" table — too heavy. The final approach: **derive personality dynamically from the existing memory table, no new tables needed.** Instead of storing a separate "personality score," the system checks what memories the agent has before each conversation and calculates how its personality should be adjusted on the fly.

```javascript
// lib/ops/voice-evolution.ts
async function deriveVoiceModifiers(sb, agentId) {
  // aggregate this agent's memory distribution
  const stats = await aggregateMemoryStats(sb, agentId);

  const modifiers = [];

  // rule-driven (not LLM)
  if (stats.lesson_count > 10 && stats.tags.includes('engagement')) {
    modifiers.push('Reference what works in engagement when relevant');
  }
  if (stats.pattern_count > 5 && stats.top_tag === 'content') {
    modifiers.push("You've developed expertise in content strategy");
  }
  if (stats.strategy_count > 8) {
    modifiers.push('You think strategically about long-term plans');
  }

  return modifiers.slice(0, 3);  // max 3
}
```

Why rule-driven instead of LLM?

1. **Deterministic**: Rules produce predictable results. No LLM hallucination causing sudden personality shifts.
2. **Cost**: $0. No additional LLM calls.
3. **Debuggable**: When a rule misfires, it's easy to track down.

Injection Method

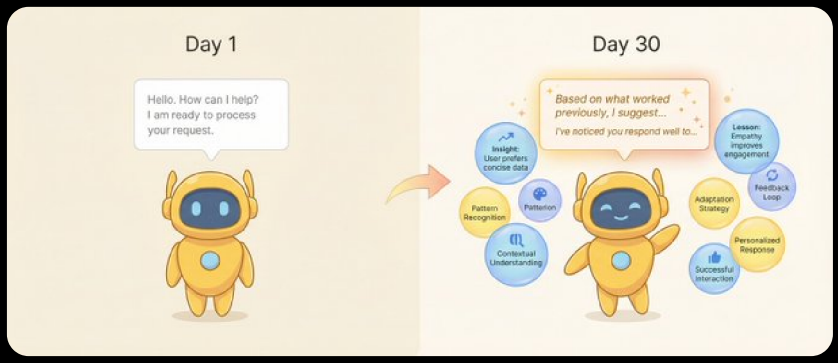
Modifiers get injected into the agent's system prompt before a conversation starts:

```javascript
async function buildAgentPrompt(agentId, baseVoice) {
  const modifiers = await deriveVoiceModifiers(sb, agentId);

  let prompt = baseVoice.systemDirective;  // base voice
  if (modifiers.length > 0) {
    prompt += '\n\nPersonality evolution:\n';
    prompt += modifiers.map(m => `- ${m}`).join('\n');
  }
  return prompt;
}
```

The effect: say your social media agent has accumulated 15 lessons about tweet engagement. Its prompt now includes "Reference what works in engagement when relevant" — and it'll naturally bring up engagement strategies in conversations.

Within the same conversation, each agent's voice modifiers are derived once and cached — no re-querying every turn.



Chapter 7: Making It Look Cool — The Frontend

> Your backend can be humming perfectly, but if nobody can see it, it might as well not exist.

The Stage Page

This is the system's main dashboard. It started as a 1500+ line mega-component — slow to load, one error meant a full white screen.

The split:

- StageHeader.tsx → Title bar + view toggle
- SignalFeed.tsx → Real-time signal feed (virtualized)
- MissionsList.tsx → Mission list + expand/collapse

- StageFilters.tsx → Filter panel
- StageErrorBoundary.tsx → Error boundary + fallback UI
- StageSkeletons.tsx → Skeleton loading screens

Virtualization

> **Beginner tip:** What's virtualization? Say you have 1,000 events in a list. If the browser renders all 1,000 DOM elements at once, the page chokes. Virtualization means — only render the 20 items currently visible, and dynamically swap content as you scroll. The user feels like they're scrolling through 1,000 items, but the browser is only rendering 20.

The system generates hundreds of events daily. Render them all? Scrolling would lag. Use @tanstack/react-virtual:

```javascript
import { useVirtualizer } from '@tanstack/react-virtual';

const virtualizer = useVirtualizer({
  count: items.length,
  getScrollElement: () => parentRef.current,
  estimateSize: () => 72,  // estimated row height
  overscan: 8,             // render 8 extra rows as buffer
});
```

500+ events, buttery smooth scrolling.

Error Boundaries

Using react-error-boundary — if one component crashes, it won't take down the whole page:

```javascript
<ErrorBoundary fallback={<p>Something went wrong. <button onClick={retry}>Re
  <OfficeRoom />
</ErrorBoundary>
```

OfficeRoom: The Pixel Art Office

This is the most recognizable piece — 6 pixel-art agents in a cyberpunk office:

- Behavior states: working / chatting / grabbing coffee / celebrating / walking around
- Sky changes: day / dusk / night (synced to real time)
- Whiteboard displays live OPS metrics
- Agents walk around (walking animations)

This component is visual candy — it doesn't affect system logic, but it's the first thing that hooks users.

Mission Playback

Click on a mission and replay its execution like a video:

```javascript
function MissionPlayback({ mission }) {
  const [step, setStep] = useState(0);
  const [playing, setPlaying] = useState(false);

  useEffect(() => {
    if (playing && step < mission.events.length - 1) {
      const timer = setTimeout(() => setStep(s => s + 1), 2000);
      return () => clearTimeout(timer);
    }
  }, [playing, step]);

  return (
    <div>
      <Timeline events={mission.events} current={step} onClick={setStep} />
      <PlayButton playing={playing} onClick={() => setPlaying(!playing)} />
      <StepDetail event={mission.events[step]} />
    </div>
  );
}
```

> **Beginner tip:** The frontend is optional. You can absolutely debug the whole system by just looking at data in the Supabase dashboard. But if you want other people to see what your agents are up to, a good-looking frontend is essential.

Chapter 8: The Launch Checklist

> You've read all 7 chapters. Here's your checklist.

Database Migrations

> **Beginner tip:** What's a migration? It's "version control for your database." Every time you create a table or change a column, you write a numbered SQL file (001, 002...). That way, anyone who gets your code can run them in order and end up with the exact same database structure.

Run your SQL migrations in this order:

001-010: Core tables (proposals, missions, steps, events, policy, memory...)
011: trigger_rules (trigger rules table)
012: agent_reactions (reaction queue table)
013: roundtable_queue (conversation queue table)
014: dynamic_relationships (dynamic relationships table)
015: initiative_queue (initiative queue table)

> **Beginner tip:** If you're using Supabase, go to Dashboard → SQL Editor and paste in the SQL. Or use the Supabase CLI: supabase db push.

Seed Scripts

Initialize data (must run after tables are created):

```bash
# 1. Core policies
node scripts/go-live/seed-ops-policy.mjs

# 2. Trigger rules (4 reactive + 7 proactive)
node scripts/go-live/seed-trigger-rules.mjs
node scripts/go-live/seed-proactive-triggers.mjs

# 3. Roundtable policies
node scripts/go-live/seed-roundtable-policy.mjs

# 4. Initial relationship data (15 pairs)
node scripts/go-live/seed-relationships.mjs
```

Key Policy Configuration

At minimum, set these:

⚙ auto_approve
→ Suggested: { "enabled": true }
→ Purpose: Enable auto-approval

⚙ x_daily_quota
→ Suggested: { "limit": 5 }
→ Purpose: Daily tweet limit (start conservative)

⚙ roundtable_policy
→ Suggested: { "enabled": true, "max_daily_conversations": 5 }
→ Purpose: Conversation cap (start conservative)

⚙ memory_influence_policy
→ Suggested: { "enabled": true, "probability": 0.3 }
→ Purpose: Memory influence probability

⚙ relationship_drift_policy
→ Suggested: { "enabled": true, "max_drift": 0.03 }
→ Purpose: Max relationship drift

⚙ initiative_policy
→ Suggested: { "enabled": false }
→ Purpose: Keep off until the system is stable

Recommendation: New features start with enabled: false. Turn them on one by one once the system is running smoothly.

How Workers Actually Execute Steps

In Chapter 1 you learned that heartbeat creates missions with steps. But how does a worker on the VPS actually pick up and execute a step? Every worker follows the same pattern:

```javascript
```

```
// The universal worker loop
async function main() {
  while (true) {
    try {
      // 1. Check if this worker is enabled (via ops_policy)
      // 2. Check quotas (e.g., daily tweet limit)
      // 3. Fetch next queued step for this kind
      const step = await sb.from('ops_mission_steps')
        .select('*')
        .eq('status', 'queued')
        .eq('kind', 'post_tweet')  // each worker handles its own kind
        .order('created_at', { ascending: true })
        .limit(1);

      if (!step) { await sleep(POLL_INTERVAL); continue; }

      // 4. Atomically claim the step (compare-and-swap)
      const { data } = await sb.from('ops_mission_steps')
        .update({ status: 'running', reserved_by: WORKER_ID })
        .eq('id', step.id)
        .eq('status', 'queued')    // only if STILL queued — this is the ato
        .select('id')
        .maybeSingle();

      if (!data) { await sleep(POLL_INTERVAL); continue; }  // another worke

      // 5. Execute the actual work
      // 6. Mark step succeeded or failed
      // 7. If all steps in the mission are done → finalize mission
    } catch (err) {
      console.error(err);
    }
    await sleep(POLL_INTERVAL);
  }
}
```

The key is step 4 — **atomic claiming**. Two workers might see the same queued step at the same time. The .eq('status', 'queued') in the UPDATE ensures only one succeeds. The loser gets zero rows back and moves on. No duplicate work.

Workers also have a **circuit breaker**: if a worker fails 3 times in a row, it auto-disables itself (sets enabled: false in ops_policy) and fires an alert. You'll see it in the dashboard rather than discovering it's broken a week later.

> **Beginner tip:** "Atomic claiming" sounds complex, but it's just a database trick. Think of it like two people reaching for the last donut — the database ensures only one hand gets it. You don't need a separate locking service; PostgreSQL handles it natively.

Environment Variables

Every worker needs these:

```
plaintext
# Required — every worker needs these
SUPABASE_URL=https://your-project.supabase.co
SUPABASE_SERVICE_ROLE_KEY=eyJ...

# Heartbeat auth (for the Vercel cron endpoint)
CRON_SECRET=your-secret-here

# Optional — with sensible defaults
WORKER_ID=worker-1              # identifies this worker in logs
POLL_INTERVAL_MS=15000          # how often to check for work (ms)
```

Store them in a .env file on the VPS with chmod 600 (owner-only read). Never put secrets in crontab or command-line arguments — they show up in process lists.

VPS Worker Deployment

One worker process per step kind:

⚡ roundtable-worker → Conversation orchestration + memory extraction + Initiative
⚡ x-autopost → Tweet publishing
⚡ analyze-worker → Analysis task execution
⚡ content-worker → Content creation
⚡ crawl-worker → Web crawling

Manage with systemd (auto-restart on crash):

```
[Service]
Type=simple
ExecStart=/usr/bin/node /path/to/worker.mjs
Restart=always
RestartSec=10
```

> **Beginner tip:** What's systemd? Linux's built-in "process nanny." You tell it "run this program," and it watches over it — process crashed? Auto-restart in 10 seconds. Server rebooted? Starts it automatically. No more waking up at 3 AM to manually restart things. If you're not familiar with systemd, give this config and your worker path to your AI coding assistant and ask it to generate the complete service file.

Verification Steps

1. npm run build — zero errors
2. Heartbeat succeeding every 5 minutes (check the ops_action_runs table)
3. Triggers are firing (check ops_trigger_rules for fire_count)
4. Roundtable conversations are running (check ops_roundtable_queue for rows with succeeded status)
5. Events are flowing (check ops_agent_events for new rows)
6. Memories are being written (check ops_agent_memory for new rows)
7. Frontend shows the signal feed (open the /stage page)

The Second Track: OpenClaw

The workers above are **reactive** — they wait for heartbeat to create missions, then poll Supabase and execute. But there's a second track.

OpenClaw is a multi-agent gateway that runs on the same VPS. It doesn't manage the workers — it runs its own **scheduled agent jobs** independently:

- Deep research cycles (agents autonomously research topics on a schedule)

- Social intelligence scans (periodic Twitter/web analysis)
- Daily briefings (agents summarize what happened)
- Memory integration (agents consolidate what they've learned)

Think of it this way:

- **Workers** = employees waiting for assignments (heartbeat assigns, they execute)
- **OpenClaw** = employees with their own daily routines (research at 9am, social scan at 1pm, briefing at 8pm — no one tells them to, they just do it on schedule)

OpenClaw's output gets bridged to your /stage frontend via a lightweight exporter script — so everything the agents do autonomously also shows up in the dashboard.

You don't need OpenClaw to get started. The heartbeat + workers loop from this tutorial is a complete system on its own. OpenClaw adds a second layer of autonomous behavior when you're ready for it.

When you are ready, I made a **Claude Code skill** for OpenClaw — install it, and your AI assistant knows how to set up and operate everything out of the box:

```bash
# Install the OpenClaw operations skill
claude install-skill https://github.com/Heyvhuang/ship-faster/tree/main/skil
```

Once installed, just tell your AI:

> "Set up OpenClaw on my VPS. Configure scheduled jobs for research, social scanning, and daily briefings."

The skill contains the complete operations reference — setup, config, cron jobs, troubleshooting. Your AI reads it and handles the rest. Trust your AI.

> **Beginner tip:** What's a Claude Code skill? It's a knowledge pack you install into your AI coding assistant. Think of it like a "cheat sheet" — except your AI reads it automatically and knows exactly what commands to run. You don't need to memorize anything.

Cost Breakdown

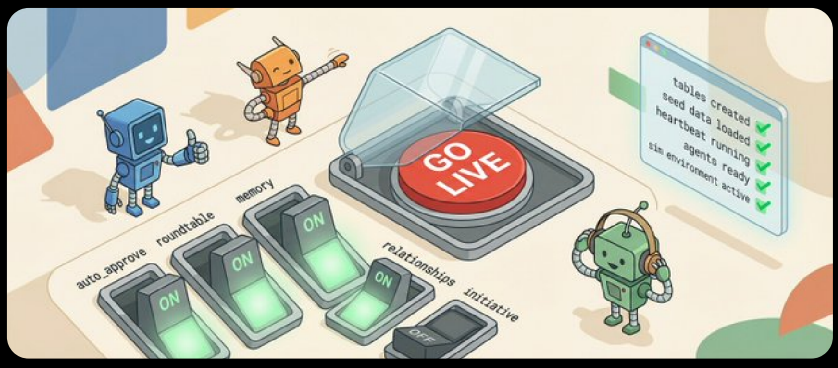
💰 LLM (Claude API) → Usage-based, ~$10-20/month
💰 VPS (Hetzner 2-core 4GB) → $8 fixed
💰 Vercel → $0 (Hobby free tier)
💰 Supabase → $0 (Free tier)

📊 **Total: $8 fixed + LLM usage**

LLM cost depends on how many conversations and tasks you run. If you stick to 3 agents + 3-5 conversations per day, you can keep LLM costs under

$5/month. Hetzner's VPS is way cheaper than AWS/GCP, and the performance is more than enough.

> **Beginner tip:** LLM APIs are "pay-per-use" — each API call costs a small amount. No calls, no charges. It's like a phone plan — you pay for what you use. Vercel and Supabase both have free tiers that are plenty for personal projects.



Final Thoughts

This system isn't perfect.

- Agent "free will" is mostly probabilistic uncertainty simulation, not true reasoning
- The memory system is structured knowledge extraction, not genuine "understanding"
- Relationship drift is small (±0.03) — it takes a long time to see significant changes

But the system genuinely runs, and genuinely doesn't need babysitting. 6 agents hold 10+ meetings a day, post tweets, write content, and learn from each other. Sometimes they even "argue" over disagreements — and the next day their affinity actually drops a tiny bit.

You Don't Have to Do Everything at Once

This tutorial is 8 chapters long. That looks like a lot. But you **really don't need to tackle it all in one go**.

Minimum viable version: 3 agents (coordinator, executor, observer) + 4 core tables + heartbeat + 1 worker. That's enough for a working loop.

Intermediate version: Add roundtable conversations + the memory system. Now agents start feeling like they're actually collaborating.

Full version: All 8 chapters. Dynamic relationships, initiative, voice evolution. Agents start feeling like a real team.

Each step is independent. Get one working before adding the next. Don't bite off more than you can chew.

Still Lost?

Paste this article into Claude and say:

> "I want to build a multi-agent system for [your domain] following this tutorial, starting with 3 agents. Generate the complete code for Chapter 1 — 4 Supabase tables + proposal-service + heartbeat route."

It'll write the code for you. Seriously.

If you build your own version following this tutorial — even if it's just 2 agents having conversations — come tell me at @Voxyz_ai.

You can see all 6 agents operating in real time at voxyz.space.

Solo devs building multi-agent systems — every person you talk to is one

Vox
@Voxyz_ai

Follow

I bridge visual AI and coding tools. Builder. Workflows. Honest takes. I share everything here ↓