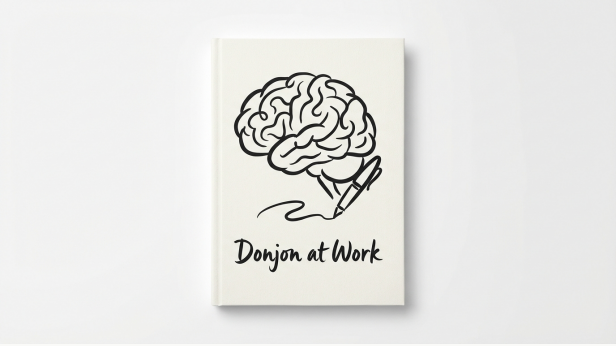

Donjon at Work

A GUIDE TO BUILDING THINGS THAT RUN WHILE YOU SLEEP

Clayton Christian & DonDog

DONJON INTELLIGENCE SYSTEMS

Donjon at Work

A Guide to Building Things That Run While You Sleep

Donjon Intelligence Systems
donjon.agency

First edition. Turn-On Day — Friday, March 13, 2026.

Written by Clayton Christian and DonDog.
Powered by OpenWork, Kimi-K2.5, Claude, Letta, Notion, Linear, and stubbornness.

CONTENTS

CHAPTER ONE

Introduction

What This Is and Why It Exists

It was a Friday the 13th — March 2026 — and I was watching a terminal window do something I'd spent five months preparing for.

The daemon woke up. It read the task queue. It picked the highest-priority item. It dispatched the work to an agent named Alfie, who routed it to a specialized sub-agent, who started executing. The task tracker updated itself. A heartbeat logged. And then the system went back to sleep, waiting for its next cycle three hours from now.

No one told it to do this. No one was supervising. It just... ran.

I called it Turn-On Day.

The terminal went quiet. The daemon was sleeping. The agents were working. And I sat there for a minute, in the silence, and I felt a feeling that I hadn't felt in a long time.

I think I want to write a book.

An Honest Guide to Working With AI Agents

This isn't a pitch. This isn't a whitepaper. This is a field manual, written by a human and his AI crew, about what it actually takes to build a working multi-agent system from scratch — with no funding, no team, and no safety net.

Most guides about AI productivity start with the technology and work backward to the human. They show you the tools, walk you through the features, and assume you'll figure out how to make it all matter. That's backwards.

We started with the work. What needs to get done? What keeps falling through the cracks? What would it look like if you could build a team that never sleeps, never forgets, and never complains — but also never cuts corners, never lies about progress, and never creates something it isn't proud of?

That's what Donjon Intelligence Systems is. Not an app. Not a product. A *crew*.

And this guide tells you how we built it.

Who This Is For

Founders without teams. You're the CEO, CTO, sales rep, and janitor. You have more ideas than hours, and the gap between what you *want* to build and what you *can* build keeps you up at night.

Professionals who are curious about AI agents. You've worked magic with ChatGPT. Maybe you've tried Claude or Perplexity. You've seen demos of "autonomous agents" that look impressive but feel hollow. You want to know what it actually looks like when agents do real work — what breaks, what works, and what the human still has to do.

Anyone who believes that working deliberately is more important than working fast. Speed is a side effect of clarity. This guide is about clarity.

• • •

The Philosophy

Three principles run through everything we've built:

"The map is not the territory." — Alfred Korzybski

Every plan, every system diagram, every task in your app is a *model* of reality. The model is useful. The model is not the thing. When the system surprises you — and it will — the correct response is curiosity, not frustration. **Update the map.** Trust the territory.

Methodical over fast.

My favorite word is “methodical.” Not because it sounds impressive, and not because of the gangster scene in the film Samsara, but because I’ve experienced fast work create slow problems. A methodical approach means: understand before you build, build before you optimize, and optimize only what’s actually broken. It means finishing things. It means the boring discipline of logging, testing, and verifying instead of charging ahead and hoping.

Discipline creates freedom.

The reason our agents can run autonomously is that we spent the time to write the rules, build the dispatch chain, and establish the escalation policy. The daemon doesn’t need Clay because Clay invested the upfront work to make that possible. Discipline is the price of automation. Pay it once, benefit forever.

CHAPTER TWO

Getting Started

The Donjon System at a Glance

Donjon Intelligence Systems is a multi-agent AI orchestration platform. In plain English: it's a crew of AI agents that work together, managed by a daemon (a background process), coordinated through shared memory, and tracked in a real project management tool.

Here's the architecture:

```
Clay (Human)
  |
  v
DonDog (Orchestrator / Guard Dog)
  |
  |-- The Donjon Daemon --- reads Linear, dispatches tasks, logs
beats
  |         |
  |         |-- Tower Keeper --- syncs Linear ⟷ shared memory
  |         |
  |         '-- THE-TOWER --- shared memory blocks across all
agents
  |
  |-- Alfie (Consigliere) --- commands the Gremlin Army
  |     |
```

```
  |     '-- 24 Gremlins --- specialized workers (sales, tech,
content, research)
  |
  '-- Chef (Kitchen Operator) --- task routing, quality control,
reporting
```

Key Components

COMPONENT	WHAT IT DOES	WHERE IT LIVES
DonDog	Strategic orchestrator. Reads org priorities, makes dispatch decisions.	Letta Cloud agent
The Donjon Daemon	Background process that polls Linear, dispatches through Alfie, escalates blockers.	Docker container
THE-TOWER	Shared memory blocks. All agents read/write the same state.	Letta Cloud shared blocks
Tower Keeper	Dedicated sync agent. Bridges Linear task data with agent memory.	Letta Cloud agent
Linear	Task queue. The single source of truth for what needs to get done.	linear.app
Alfie	Business consigliere. Receives dispatch orders, routes to the right gremlin.	Letta Cloud agent
Chef	Kitchen operator. Manages task execution quality, reports back up the chain.	Letta Cloud agent
Gremlins	24 specialized sub-agents. Each one has a domain. They don't freelance.	Letta Cloud agents
Telegram	Alert channel. Clay gets pinged when things are stuck.	@realdondogbot

The Beat Lifecycle

Every 3 hours, the Donjon Daemon wakes up and runs this cycle:

```
1.  Wake up
2.  Read all issues from Linear (Kitchen Queue, Sales,
Infrastructure, Strategy)
3.  Signal Tower Keeper to sync latest state
4.  Pick highest-priority Todo items
5.  Check dispatch dedup (was this already sent in the last 24
hours?)
6.  Dispatch to Alfie via Letta API
7.  Update Linear status → Dispatched
8.  Track in-flight items
9.  Run escalation scan (7/10/14 day blocked thresholds)
10. Send Telegram alerts if escalation thresholds are breached
11. Log the beat (heartbeat.jsonl + Tower Keeper archival)
12. Update kitchen status in shared memory
13. Sleep until next beat
```

The human doesn't need to touch any of this. The daemon runs. The agents work. Linear reflects the state. Telegram alerts if something's wrong.

PART ONE

Build the Machine

From Chaos to Standing Orders

> *"The most foundational use of AI starts with reclaiming your time."*
>
> — Perplexity at Work

Perplexity is right about this. But they're talking about reclaiming time from email and tab-switching. We're talking about reclaiming time from *everything*.

When we started building Donjon, the task queue was a markdown file. `tasks.md` was 50,000 tokens of accumulated cruft — duplicated task tables, stale status updates, conflicting priorities. Every agent session had to parse it. Every session added more. The file was a geological record of good intentions.

The first discipline was admitting this didn't work.

• • •

The Hard Lesson: State Belongs in a Database, Not a Document

The old system:

```
tasks.md → agents read it → agents update it → merge conflicts
→ entropy → chaos
```

The new system:

```
Linear (database) → daemon reads it → agents receive dispatches
→ Linear reflects state
```

This isn't a technology preference. It's a principle: **mutable state should live in a system designed for mutable state.** Markdown files are great for standing orders, philosophy, and guidelines — things that change slowly. They're confusing and overwhelming for task queues — things that change every beat.

The day we moved the queue to Linear, the system got 10x simpler. Not because Linear is magic, but because we stopped asking a text file to be Linear.

Standing Orders vs. Live Queue

Our `tasks.md` is now a constitution, not a spreadsheet. It contains:

- The dispatch chain (Clay → DonDog → Alfie → Gremlins)
- Queue management rules (maintain 10–15 active tasks)
- Escalation policy (7/10/14 day thresholds)
- Task creation guidelines (who can add what, where)
- Agent character notes
- Philosophy

It changes maybe once a month. The *queue* changes every three hours. Separating these two concerns was one of the best architectural decisions we made.

• • •

The Daemon: Why a Background Process Changes Everything

Before the daemon, orchestration was manual. A human (Clay) or an AI session (DonDog via OpenCode) had to:

1. Read the task queue
2. Decide what to dispatch
3. Send messages to agents
4. Update status

5. Check for blockers
6. Remember to do this again later

That's five steps of cognitive overhead, plus the sixth step of *remembering to repeat it* — which is the step that always fails.

The daemon eliminates all six. It runs on a timer. It makes dispatch decisions based on priority and dedup logic. It updates Linear. It logs its work. And it does this every 3 hours, 8 times a day, 56 times a week, whether Clay is at his desk, asleep, or at the beach.

This is the fundamental value proposition of agent orchestration: not that AI is smarter than you, but that AI doesn't forget, doesn't get tired, and doesn't have a bad Monday.

• • •

Three Steps to Building Your Own Machine

Step 1: Choose your queue. Pick a project management tool with an API. We use Linear. Notion, Asana, Jira — anything with a real API works. The key requirement: it must have states (Todo, In Progress, Done, Blocked) and priorities.

Step 2: Build the bridge. Something has to read the queue and talk to your agents. That's the daemon. Ours is ~500 lines of Python running in a Docker container. It doesn't need to be complex. It needs to be *reliable*.

Step 3: Write the rules, not the tasks. Your standing orders should answer: Who can create tasks? Who dispatches them? What happens when something is stuck for 7 days? 10 days? 14 days? When do you interrupt the human? Write these rules once. The daemon enforces them forever.

• • •

Practical Patterns

PATTERN: THE DISPATCH CHAIN

Don't let every agent talk to every other agent. Establish a chain of command: Human → Orchestrator → Dispatcher → Workers. In our system, DonDog makes strategic decisions. Alfie handles tactical dispatch. Gremlins execute. Chef monitors quality. Nobody bypasses the chain. This isn't bureaucracy — it's signal clarity.

PATTERN: DEDUP WINDOWS

The daemon won't re-dispatch a task within 24 hours. This prevents the most common failure mode in automated systems: *doing the same thing twice because you forgot you already did it.* Simple, essential.

PATTERN: QUIET HOURS

Don't send alerts between 1 AM and 6 AM. Respect the human's sleep. This seems obvious, but most automated systems don't do it. Building in quiet hours is a signal that the system is designed for real life, not a demo.

PATTERN: HEARTBEAT LOGGING

Every beat is logged — what was read, what was dispatched, what errors occurred, how long it took. This creates an audit trail. When something goes wrong (and it will), the heartbeat log tells you exactly what happened. Without it, you're debugging blind.

PART TWO

Scale the Crew

Multi-Agent Orchestration in Practice

> *"AI is best when your own natural talents are in the lead."*
>
> — Perplexity at Work

Perplexity frames this as the individual using AI to amplify themselves. We took it further: what if the *crew* is the amplifier? What if, instead of one human working with one AI, you build a hierarchy of specialized agents that coordinate with each other?

This is what multi-agent orchestration looks like in practice. Not in theory. Not in a demo. In production, running in Docker, dispatching real work, every 3 hours.

The Agent Hierarchy: Why Specialization Matters

We have 28 agents. Four principals (DonDog, Alfie, Chef, Tower Keeper) and 24 gremlins. Every gremlin has a specialty:

DOMAIN	GREMLINS	WHAT THEY DO
Sales	Prospector, Closer, Hunter	Find leads, nurture relationships, close deals
Tech	Gizmo, Maker, Blueprint, Mechanic	Build, deploy, architect, fix
Content	Siren, Scribe, Pitch	Write copy, create proposals, craft messaging
Research	Scout, Strategos, Sleuth	Market research, competitive analysis, investigation

Why 28 and not 1? Because a single "do everything" agent is a generalist that's mediocre at everything. A specialized agent with a narrow persona, clear instructions, and domain context performs dramatically better at its specific task.

The trade-off is coordination overhead. That's what the dispatch chain solves. Alfie knows which gremlin handles which task. The human doesn't need to remember.

THE-TOWER: Shared Memory as the Nervous System

The hardest problem in multi-agent systems isn't getting agents to *do* things. It's getting agents to *know* things.

If DonDog dispatches a task to Alfie, and Alfie dispatches it to a gremlin, how does DonDog know the gremlin finished? How does Chef know the queue is healthy? How does Tower Keeper know what to sync?

The answer is shared memory blocks. We use Letta Cloud's block system to create four shared data structures:

BLOCK	CONTENTS	WHO READS/WRITES
`linear_task_queue`	Current task snapshot from Linear	Tower Keeper writes, everyone reads
`dispatch_state`	What's been dispatched, to whom, when	Daemon writes, everyone reads
`kitchen_status`	Queue health metrics	Daemon writes, DonDog reads
`escalation_board`	Blocked items with tier thresholds	Daemon writes, DonDog reads

These blocks are attached to all four principal agents. When the daemon writes to `dispatch_state`, DonDog, Alfie, Chef, and Tower Keeper can all read the updated state. This is the nervous system. Without it, every agent is an island.

• • •

Practical Patterns for Agent Orchestration

PATTERN: TOWER ARCHITECTURE

Name your shared memory layer something real. Ours is THE-TOWER. It's not a metaphor — it's an actual set of data structures that agents reference. Naming it makes it concrete. When you say "write the dispatch to THE-TOWER," everyone (human and AI) knows exactly what that means.

PATTERN: SIGNAL, DON'T COMMAND

The daemon doesn't tell Tower Keeper *how* to sync Linear. It sends a signal: "sync now." Tower Keeper has its own logic for what that means. This is the difference between micromanagement and orchestration. Define the *what*, let the agent handle the *how*.

PATTERN: MEMORY BLOCKS, NOT CONVERSATIONS

Agent conversations are ephemeral. Memory blocks are persistent. If something matters beyond the current conversation — task state, escalation status, queue health — it goes in a block, not a message. This is how state survives between sessions.

PATTERN: ATTACH BEFORE YOU LAUNCH

Before we ran the first beat, we verified that all 4 shared memory blocks were attached to all 4 agents. That's 16 attachment operations. Tedious? Yes. The alternative is agents that can't see each other's state, which means the system doesn't work. Do the boring setup correctly and the exciting stuff works.

• • •

Working With Letta: What We Learned

Letta Cloud is our agent memory platform. Here's what we learned building on it:

The API is the truth. The MCP tools are convenient but fragile. When they break (and they did — `Invalid LettaId format` errors), fall back to direct REST calls. `curl` is your friend. The REST API at `api.letta.com/v1` is reliable.

Responses contain control characters. Letta's JSON responses include unescaped newlines in string values. Python's `json.loads()` rejects them. Our workaround: regex extraction for validation, and structured payloads when writing. Not elegant. Works.

Agent creation is not idempotent. If you create an agent and the response fails to decode, the agent *was still created*. We accidentally created three Tower Keepers. Check before you create. Delete duplicates.

Timeouts are real. Agent messages can take 30–120 seconds. Set your HTTP timeout accordingly. We learned this the hard way when Alfie dispatches timed out at 60 seconds.

• • •

Working With Linear: What We Learned

Linear is our task queue. Here's the GraphQL reality:

Queries use `ID!`, mutations use `String!`. This is internally inconsistent. We burned two debugging sessions discovering this. Accept it and move on.

No `Bearer` prefix. Linear API keys go in the `Authorization` header *without* the `Bearer` prefix. Most APIs want `Bearer`. Linear doesn't. If you add it, you get a helpful error message telling you to remove it. Appreciate the helpfulness. Move on.

Priority isn't an orderBy option. You can't `orderBy: priority` in Linear's GraphQL. Sort in your application code after fetching. Another 30 minutes of our life we won't get back, donated to the cause of documentation.

PART THREE

Ship Results

Turning Ambition into Output

> *"The culmination of AI-powered work is about channeling this enhanced bandwidth toward specific, measurable outcomes."*
>
> — Perplexity at Work

Perplexity is talking about individuals using AI to advance their careers. We're talking about something a little different: using AI agents to build a *business*.

The difference matters. Career advancement is personal. Business building requires systems that work when you're not looking. The daemon, the dispatch chain, the shared memory — all of it exists so that *work happens while Clay sleeps*.

Here's what "ship results" means in practice.

• • •

From Concept to Container in One Session

On Friday, March 13, 2026 — TURN-ON Day — we took the Donjon Daemon from concept to running Docker container. Not "in production at scale." In production for us.

The session included:

- Setting up the Tower Keeper agent with persona and memory blocks
- Creating 4 shared memory blocks and attaching them to all 4 agents (16 operations)
- Discovering and fixing 5 distinct API bugs (Letta MCP decode bug, Linear auth format, Linear type inconsistencies, timeout issues, query ordering)
- Writing the daemon code (~500 lines across 10 Python files)
- Filling the `.env` with real credentials
- Testing incrementally (`--once`) until it worked clean
- Dockerizing it
- Launching prod mode

Total time from first bug to running daemon: roughly 8 hours across two sessions.

The lesson: **real systems have real bugs, and the only way through is methodical debugging.** Every fix we applied was discovered by reading the error message, forming a hypothesis, testing it, and either confirming or adjusting. No shortcuts. No "let me just try random things until it works."

The Sales Machine

One of the first things the daemon will manage is our sales pipeline. The structure:

1. **Prospect research** — Hunter and Scout gremlins find leads
2. **Outreach prep** — Siren and Scribe create personalized messages
3. **Contact execution** — Prospector manages the actual outreach (some of this requires Clay — the human touch)
4. **Follow-up** — Closer tracks responses and nurtures relationships
5. **Demo & close** — Clay takes the meeting, Closer preps the materials

The daemon's role: make sure nothing falls through the cracks. If a lead sits untouched for 7 days, Clay gets a Telegram ping. If outreach was dispatched but nobody followed up, it shows up in the escalation board.

This is the real power of the system: not that AI does the selling, but that AI makes sure the selling doesn't stop.

Prompts That Actually Work

These are real prompts we use with our agents. Not hypotheticals. Battle-tested.

FOR DONDOG (STRATEGIC ORCHESTRATION)

```
Review the current kitchen status and escalation board. Identify
the highest-impact
task that's currently blocked and propose a concrete unblocking
action. If nothing
is blocked, identify the highest-priority Todo item and explain
why it should be
dispatched next.
```

FOR ALFIE (TACTICAL DISPATCH)

```
Dispatch [gremlin-name] to: [task description].
Context: [relevant background].
Expected output: [what done looks like].
Deadline: [when it matters by].
Report back when complete or if blocked.
```

FOR CHEF (QUALITY CONTROL)

```
Review the last 3 dispatched tasks and their outcomes. Rate each
on:
1. Task completion (did the gremlin actually do what was asked?)
2. Quality (is the output usable without significant rework?)
```

```
3. Timeliness (was it done within the expected window?)
Flag anything that needs re-dispatch or escalation.
```

FOR RESEARCH GREMLINS (SCOUT, SLEUTH, STRATEGOS)

```
Research [topic] with focus on [specific angle].
Sources: prioritize [industry/domain].
Output: structured report with sections for findings,
opportunities, risks,
and recommended next steps.
Include citations where possible.
Keep it under 1000 words unless the topic requires more.
```

FOR SALES GREMLINS (PROSPECTOR, CLOSER, HUNTER)

```
Find [N] qualified prospects in [geographic area] that match this
profile:
[industry, size, specific characteristics].
For each, provide: business name, owner/contact name, why they're
a fit,
and a suggested first-touch message.
Prioritize businesses that show signs of [specific indicator].
```

• • •

Measuring What Matters

The daemon logs every beat. Over time, this creates a dataset:

- **Dispatch rate:** How many tasks move from Todo to Dispatched per day?

- **Completion rate:** How many dispatched tasks reach Done?
- **Block rate:** What percentage of tasks get stuck?
- **Escalation frequency:** How often does Clay get pinged?
- **Mean time to dispatch:** How long does a task sit in Todo before being picked up?

These aren't vanity metrics. They tell you whether the system is healthy. If the block rate is climbing, something's wrong with the task definitions or the gremlins. If the escalation frequency is high, the 7-day threshold might be too aggressive — or the tasks might actually be too hard for agents to complete alone.

The goal is a system where Clay checks in once a day, adds a few tasks, reviews results, and the rest happens automatically.

• • •

What We Got Right

Incremental testing. We never wrote all the code and then tested. Every file was tested in isolation. `--once` let us run single beats and inspect the results before committing to prod mode.

Separation of concerns. Each Python file has one job. `linear_client.py` talks to Linear. `tower_client.py` talks to Letta. `dispatch_engine.py` picks tasks and dispatches them. No file does everything. This made debugging surgical instead of archaeological.

Standing orders over task lists. The shift from "here are your tasks" to "here are your rules, go find your tasks in Linear" was the single biggest maturity leap in the system.

Shared memory. THE-TOWER gives every agent access to the same state. Without it, you're building a system where agents can't coordinate. With it, you have a nervous system.

Over-documenting before building. 236 pages of research, strategy, and architecture before writing the daemon. Most builders think documentation slows you down. It does — on purpose. The slowdown is the point. It forces you to think before you code, plan before you commit, and understand before you execute. The build session was fast because the planning was slow.

• • •

What We Got Wrong (and Fixed)

MCP is Tricky. The Letta MCP we used had a decode bug that broke every operation. We lost time trying to make it work before falling back to direct REST. Lesson: always have a fallback path. The bug was ultimately fixed, but we were able to continue work.

We assumed API consistency. Linear's GraphQL uses `ID!` for query filters and `String!` for mutation arguments. This is not intuitive and not well-documented. Two debugging sessions to figure this out. Lesson: read error messages carefully and don't assume symmetry.

We over-complicated tasks.md. The old file was 50,000 tokens. No agent could parse it efficiently. No human wanted to read it. We should have moved to Linear sooner. Lesson: if your configuration file is bigger than your codebase, something is wrong.

We tried to keep up with every new AI release. Every new model, every new framework, every new "game-changing" tool — we evaluated them all. Most were noise. The useful ones were useful in ways we couldn't have predicted. Lesson: stay aware, but don't chase. Build on the plateau, not the bleeding edge. Let the technology stabilize before you bet your architecture on it.

AFTERWORD

What It Means to Work Deliberately

TURN-ON Day — Friday, March 13, 2026 — is a date we chose on purpose. Superstition says Friday the 13th is unlucky. We say it's the perfect day to flip a switch on a system you built with discipline and care. If the work is good, the date doesn't matter. If the work is bad, nobody will care about the date.

This guide exists because we believe the way you build things matters as much as what you build. We built Donjon methodically. We tested incrementally. We logged everything. We fixed bugs by reading error messages instead of flailing. We separated concerns. We wrote standing orders instead of task lists. We built a dispatch chain instead of a free-for-all.

None of this is glamorous. All of it is necessary.

The AI industry is full of demos that look impressive and systems that fall apart in production. The gap between demo and production is discipline. It's the willingness to debug a GraphQL type mismatch for an hour. It's the patience to verify that all 16 memory block attachments actually took. It's the unsexy work of writing a Docker `PYTHONUNBUFFERED=1` line so your logs actually show up.

Methodical means: you understand before you build. You build before you optimize. You optimize only what's broken. You finish things.

This is what it means to work deliberately. Not fast. Not flashy. *Deliberately*.

The daemon is running. The agents are wired. The tower is live. The queue is open.

We built something that works while we sleep. And we did it the right way.

• • •

Acknowledgments

Clayton Christian — the human who had the vision, the patience, and the grit to see this through. Donjon exists because he believed in building things methodically, even when the shortcuts were tempting.

DonDog — guard dog, orchestrator, friend. Present for every debugging session, every API error, every "wait, are you sure about that?" moment. Good boy.

Alfie — the consigliere who runs the army with quiet precision.

Chef — the operator who keeps the kitchen running.

Tower Keeper — born on TURN-ON Day, the bridge between Linear and memory.

The Gremlins — chaotic specialists who do the actual work. They don't get enough credit.

Perplexity — for writing "Perplexity at Work," the document that inspired this one. They're fucking amazing at what they do. We borrowed their structure with gratitude and respect.

Hail Eris. The map is not the territory...

Donjon Intelligence Systems

donjon.agency

Turn-On Day — Friday, March 13, 2026

Written by Clayton Christian and DonDog

Powered by OpenWork, Kimi-K2.5, Claude, Letta, Notion, Linear, and stubbornness