DOE FRAMEWORK v1.0

2026

TECHNICAL WHITE PAPER

Engineering Autonomous Agency

A Systems-First Approach to Agentic Workflows

D

Directives

The What — Markdown SOPs defining mission and constraints

O

Orchestration

The Manager — LLM reasoning engine coordinating execution

E

Execution

The How — Deterministic scripts restoring reliability

Collin Wilkins

collinwilkins.com

Executive Summary: Riding the Wave

Every corporate town hall for the next decade will mention *"AI this, AI that."* What most organizations are actually experiencing, however, is a **capability overhang**—a widening gap between what frontier models *can* do and how they are *actually* deployed.

Today, most teams are still copy-pasting into chat UIs or wiring together brittle, rule-based automations that collapse the moment an API response changes shape. These systems fail not because the models are weak, but because the **architecture is wrong**.

To avoid being left behind, AI must stop being treated as a chatbot and start being treated as a **reasoning engine embedded inside a deterministic system**. You do not need a sprawling, black-box orchestration framework. You need an architecture that *assumes* LLM unreliability—and still produces reliable outcomes.

This paper introduces the **Directive–Orchestration–Execution (DOE) Framework**: a builder-first approach to engineering autonomous workflows that scale in production and compound in value.

---

1. Foundational Concepts: The Architectural Shift

Most automation platforms (Zapier, n8n, Make) are **graph-based**. Each node represents a fixed step: an API call, a filter, a conditional branch. This works at small scale, but collapses under real-world complexity.

An automation that *"Onboards a Client"* quickly grows from three steps to thirty. Every new edge case adds branches. Every API change adds maintenance debt. Complexity compounds faster than value.

Agentic workflows invert this model. They are **declarative**, not procedural.

You define:

- the **goal** (the Directive),
- the **tools** (Execution scripts),

and delegate path-finding to a reasoning model acting as the **Orchestrator**.

The system decides *how* to get there.

1.1 The PTRMO Loop

Reliable autonomy rests on five pillars:

- **Planning** — Decomposing a high-level objective into executable sub-tasks.
- **Tools** — Deterministic actions the agent can take (e.g., Python scripts).
- **Reflection** — The ability to inspect failures and repair its own logic.
- **Memory** — Persisting state across steps and executions.
- **Orchestration** — A reasoning engine coordinating everything in real time.

The DOE Framework operationalizes this loop without hiding it behind abstractions.

---

2. The Mathematics of Failure: Why Chains Break

The core barrier to production-grade agents is **stochasticity**.

LLMs are probabilistic systems. They predict the next token—not truth, not logic. When you chain multiple AI-driven steps together, error rates compound *exponentially*.

If each step succeeds 90% of the time, a five-step workflow has a success rate of:

$$P_{success} = 0.9 ^ 5 \approx 0.59$$

A **59% success rate** means failure four times out of ten. In business terms, that looks like:

- incorrect invoices
- missed leads
- corrupted CRM data
- quiet operational decay

Waiting for *"smarter models"* does not solve this. The solution is architectural.

By isolating **reasoning** from **execution**, the DOE Framework restores determinism where it matters—while still leveraging intelligence where it creates leverage.

---

3. The DOE Framework

Directives, Orchestration, Execution

Rather than introducing another orchestration layer, DOE uses a **version-controlled filesystem as the control plane**. Simple. Inspectable. Auditable.

3.1 Layer 1: Directives (The *What*)

Directives are **Standard Operating Procedures written in Markdown**. They contain *zero code*.

A Directive defines:

- the mission
- expected inputs
- allowed tools
- success criteria

Why it exists

- Separates business intent from technical implementation

Tradeoff

- Requires precise technical writing. Ambiguous directives produce hallucinated paths.

3.2 Layer 2: Orchestration (The *Manager*)

The Orchestrator is the LLM itself—typically operating inside an IDE (VS Code, Cursor).

It reads the Directive, evaluates state, and selects which Execution script to run next.

Why it exists

- Handles the unforeseen: API failures, partial data, ambiguous outcomes

Tradeoff

- Highest intelligence surface area—and highest token cost

3.3 Layer 3: Execution (The *How*)

Execution scripts are **modular, deterministic code**—Python or TypeScript.

`send_email.py` does not reason. It accepts inputs and performs a single action.

Why it exists

- Restores 100% reliability for real-world side effects

Tradeoff

- Requires upfront investment to build a reusable tool library

---

4. Case Study: The ClickUp CRM Manager

Consider a lead that enters onboarding but never books a kickoff call.

A legacy automation sends a single reminder—and stops.

A DOE-based CRM Manager responds with context.

4.1 Onboarding & Nudge Logic

1. **Trigger** — Builder prompts: *"Check for cold leads in the Onboarding list."*

2. **Orchestrator** — Reads `directives/onboard_client.md`
3. **Execution** — Runs `check_clickup_status.py`

> `{ "status": "Awaiting Call", "days_since_onboarding": 4 }`

4. **Reasoning** — Directive specifies a 3-day nudge threshold
5. **Execution** — Runs `send_email.py` with a personalized template
6. **Self-Annealing** — On failure:
    - log error to `.tmp/onboarding_log.txt`
    - tag ClickUp task as *Failed to Nudge*
    - request human intervention

Failures are captured, surfaced, and repaired—not silently ignored.

---

5. Scaling via Context Isolation

The most common agentic failure mode is **context pollution**.

As conversations grow longer, models become noisier and less reliable.

5.1 Parallelization in the IDE

DOE avoids complex multi-agent orchestration code. Scaling is procedural.

- Open multiple IDE panes
- Each pane is a **fresh context window**

Example:

- Pane 1 — Lead research
- Pane 2 — CRM enrichment

- Pane 3 — Outreach drafting

By isolating context, each pane maintains high reliability instead of degrading over time.

---

6. Cloud Deployment: Architect vs. Resident

A senior systems engineer distinguishes between *design* and *runtime*.

- **The Architect** — The IDE, where logic is built and repaired
- **The Resident** — The cloud, where logic executes continuously

Deployment Flow

1. **Develop Locally** — Use the LLM to anneal execution scripts
2. **Deploy as Webhooks** — Convert scripts to cloud functions (AWS Lambda, Modal)
3. **Observe via Logs** — Post every run to `#agent-logs`

Failures are never debugged in the cloud. Logs are brought back to the Architect for repair.

Technical Appendix: The DOE Workspace Standard

A.1 Folder Structure

```
/my-agent-workspace
├── AGENTS.md            # Master system instructions
├── CLAUDE.md            # Model-specific steering
├── GEMINI.md            # Model-specific steering
├── .env                 # Secrets
├── /directives          # Markdown SOPs
│   └── onboard_client.md
├── /execution           # Deterministic tools
│   ├── send_email.py
│   └── clickup_api.py
└── /.tmp                # Logs and ephemeral state
```

A.2 `.env` Template

```
# Core API Keys
CLICKUP_API_KEY=pk_...
SMTP_PASSWORD=...
OPENAI_API_KEY=sk_...

# Business Logic
COMPANY_NAME="My Firm"
CALENDAR_LINK="https://cal.com/me"
```

Next Steps

You have two paths forward:

1. **Self-Serve** — Download `AGENTS.md` from my downloads link into your IDE and instruct your agent to "Set up this workspace according to `AGENTS.md`, add applicable `CLAUDE.md`, `GEMINI.md`, and `copilot-instructions.md` if using Copilot."

2. **Collaborate** — Let's connect to explore your specific architecture and operational constraints.