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 massive, 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.

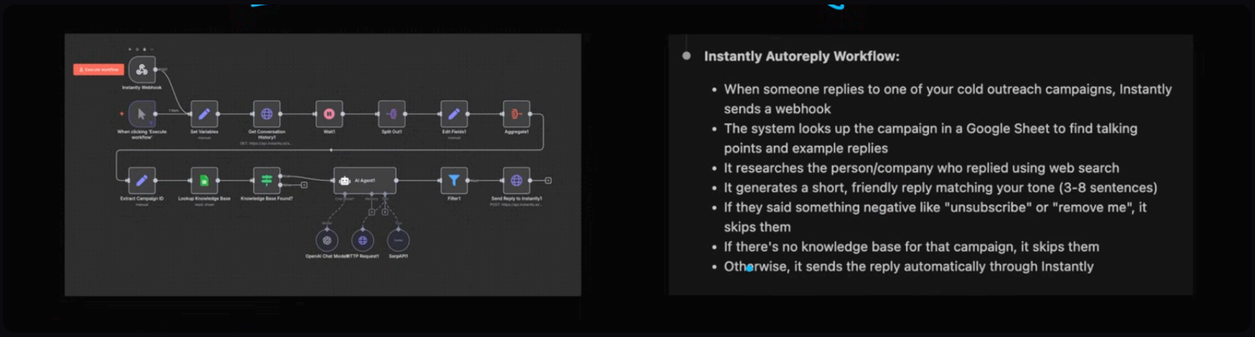
1. Foundational Concepts: The Architectural Shift

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 and people have built thousands of systems this way, but it requires specific knowledge of that platform and needs to be tinkered with for any small failures or changes.

Agentic Workflows have changed the game (automation tools & chatbots vs Agentic Workflows)

- Model intelligence just crossed a threshold
    - Frontier models have gotten really smart, they score 80% on SWE-bench verified!

- Tool integration is now standardized
    - Protocols like MCP (Model Context Protocol) standardized how ai connects to external tools and dbs
    - Other frameworks like Directive, Orchestration, Execution (DOE) and Claude Skills have formalized tool-calling
    - LLMS (and IDEs) are flexible
- Compare n8n (left) vs agentic workflows (right).



- **Instantly Autoreply Workflow:**
    - When someone replies to one of your cold outreach campaigns, Instantly sends a webhook
    - The system looks up the campaign in a Google Sheet to find talking points and example replies
    - It researches the person/company who replied using web search
    - It generates a short, friendly reply matching your tone (3-8 sentences)
    - If they said something negative like "unsubscribe" or "remove me", it skips them
    - If there's no knowledge base for that campaign, it skips them
    - Otherwise, it sends the reply automatically through Instantly

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).
    - Standardizing tools lets everyone use them regardless of model type. Also ensures consistent inputs and outputs, important for business
    - **Key Insight**: You don't need to wait for people to build them, you can build own tool from scratch. Let the model build the tools it needs
    - LLMs are probabilistic (99% so the more steps -> more variability) while Python is procedural (given inputs will get same outputs)
- **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 (covered later) 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 don't predict the single 'best' next word, they predict a distribution of options. 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:

Probability of success = 0.9 ˙ 5 · 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.

You can see the specific file structure in the Appendix A1. DOE is just a folder structure with two main folders: directives and execution; that's really all you need to get started

- Directives folder holds Standard Operating Procedure (SOP) markdown files - aka an agent's instruction manual
    - "Markdown" or .md is just a common markup language designed to format plain text; widely used in technical documentation
- Execution folder holds python (or language of choice) scripts, where the automation logic lives

WHY IT WORKS

- It is intuitive and easy to understand.
- It reduces stochasticity by separating concerns into 3 layers and (in turn) error rate.

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).

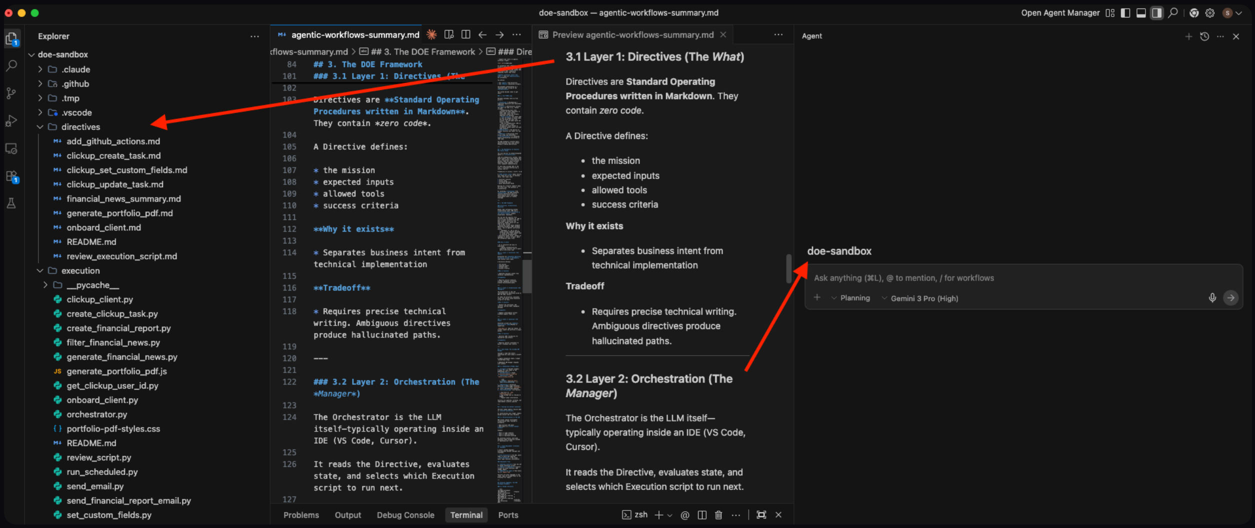
It reads the Directive, evaluates state, and selects which Execution script to run next.

Why it exists

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

Tradeoff

- Highest intelligence surface area—and highest token cost


doe-sandbox — agentic-workflows-summary.md
Open Agent Manager
Explorer
agentic-workflows-summary.md
kflows-summary.md > ## 3. The DOE Framework > ### Dire
Preview agentic-workflows-summary.md
Agent
Problems
Output
Debug Console
Terminal
Ports
zsh

```
doe-sandbox
  .claude
  .github
  .tmp
  .vscode
  directives
    add_github_actions.md
    clickup_create_task.md
    clickup_set_custom_fields.md
    clickup_update_task.md
    financial_news_summary.md
    generate_portfolio_pdf.md
    onboard_client.md
    README.md
    review_execution_script.md
  execution
    __pycache__
    clickup_client.py
    create_clickup_task.py
    create_financial_report.py
    filter_financial_news.py
    generate_financial_news.py
    generate_portfolio_pdf.js
    get_clickup_user_id.py
    onboard_client.py
    orchestrator.py
    portfolio-pdf-styles.css
    README.md
    review_script.py
    run_scheduled.py
    send_email.py
    send_financial_report_email.py
    set_custom_fields.py
```

```
## 3. The DOE Framework
### 3.1 Layer 1: Directives (The

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.
```

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.

doe-sandbox

Ask anything (⌘L), @ to mention, / for workflows

Planning   Gemini 3 Pro (High)

Notice the IDE (Layer 1 and 3 on the left in my workspace, Layer 2 **IS** the LLM in my terminal window)

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. How to Improve Agentic Workflows Over Time

5.1 The Order-of-Magnitude Rule

Only optimize when you can achieve an **order-of-magnitude improvement** in a key metric.

As a rule of thumb:

- Optimize **only if you can achieve ·5× improvement** in:
    - Time
    - Cost
    - Accuracy
    - Reliability

A workflow that runs in **2 minutes instead of 3** is rarely meaningful—unless runtime is *the* core constraint. Most of the time, it isn't.

Chasing small deltas creates the illusion of progress while consuming real engineering effort.

5.2 Beware of Micro-Optimizations

Small optimizations often come with hidden costs:

- Reduced accuracy
- Lower reliability
- Increased fragility
- Harder-to-debug systems

In agentic systems especially, shaving seconds off execution often means:

- More brittle prompts
- Tighter coupling between steps
- Less tolerance for ambiguity or edge cases

This is how systems slowly decay.

Do not overengineer for marginal gains.

5.3 Time Beats Money

In agentic workflows, **your time is the dominant constraint**.

- Compute is cheap
- Tokens are cheap
- Engineering attention is not

Spend effort where it *actually* compounds:

- Reusable abstractions
- Durable directives
- Execution patterns that generalize

If an optimization saves money but costs hours of design and maintenance, it is almost always the wrong trade.

5.4 Treat Your Library as the Asset

Your real leverage is not any single workflow—it's your **library**.

- Your prompt directives are **infinitely reusable**
- Execution scripts can be dropped into any workspace
- System instructions travel cleanly across projects

This is no different from:

- Make.com automations
- n8n workflows
- Gumloop pipelines

Same principle.

Over time, your IDE or agent workspace becomes a **treasure trove of deployable capability**.

Eventually, your system can support:

- **Automatic lead scraping**
- **Email enrichment and validation**
- **Personalized outreach and follow-ups**
- **Automated proposal generation**
- **Slide deck creation that matches your tone of voice**

And critically—

> All of it customized to communicate in *your* intent, *your* voice, and *your* standards. Not generic AI output.

---

6. Scaling via Context Isolation

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

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

6.1 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.

7. Self-Annealing Workflows

Self-annealing simply means *self-hardening*.

A self-annealing workflow:

- Diagnoses its own failures
- Applies fixes
- Updates its internal state
- Retries execution

In practice, this looks like a tight feedback loop:

> Diagnose → Fix → Update → Retry

Rather than treating errors as terminal, the agent treats them as input. Over time, this introduces **stability**, not brittleness.

The system does not need to be perfect—it needs to *learn how to recover*.

8. AI Safety in a Text-Based Interface

Agentic systems are probabilistic. That reality must be designed around, not ignored.

8.1 Guardrails for Cost and Risk

At minimum, agents should be constrained by the following rules:

- **Confirm before making API calls above a defined cost threshold**
  Example: explicit approval required for usage exceeding $5
- **Never modify credentials or API keys without explicit approval**
- **Never move secrets out of `.env` files or hardcode them into the codebase**
- **Log all self-modifications**
  - Append changes as a **changelog at the bottom of the directive**
  - Treat every mutation as auditable state

8.2 Accepting the Probabilistic Tradeoff

Agents do not offer 100% compliance. That is not a flaw—it is the cost of flexibility.

Instead of chasing perfect prevention:

- Design for **graceful recovery**
- Expect deviations
- Make failures observable and reversible

This mindset shift is foundational to reliable agentic systems.

9. Maximizing Efficiency and Throughput

Speed is not just compute—it is human interaction bandwidth.

9.1 Speak, Don’t Type

Enable dictation or voice-to-text wherever possible.

- Average typing speed: **50–70 WPM**
- Average speaking speed: **150–200 WPM**

That delta compounds quickly in long sessions.

9.2 Specificity Without Overthinking

Specificity matters—but memorization does not.

- To run a workflow, **just ask for it**
- You do not need to remember directive names
- The agent will scan for applicable instructions

Think of it like ordering food:

- You ask for the item ("I’d like a cheeseburger with fries)
- The kitchen handles the execution

9.3 Steering vs. Error Surface Area

- Vague prompts work—but expect clarifying questions
- Specific prompts take seconds longer but reduce ambiguity

Every additional step delegated to the agent introduces another chance for error.

Compounding error is real:

```
.9 × .9 × .9 ≠ .9
```

Precision early reduces correction later.

10. Watching Workflows as They Run

Do not treat workflows as black boxes.

10.1 Why Observation Matters

Watching execution in real time helps you:

- Understand internal decision paths
- Identify failure points faster
- Iterate more effectively

10.2 Use the Thinking / Reasoning View

Opening the reasoning or thinking tab is not optional—it is educational.

- You learn *how* the agent thinks
- This matters because the agent will increasingly replace manual effort
- Understanding its mental model improves delegation quality

11. Hooks, Notifications, and Long-Running Jobs

For longer workflows, passive waiting is inefficient.

Set up hooks:

- Audio chimes
- Notifications
- Status prompts when input is required

TL;DR:
If a workflow takes long enough for you to context-switch, it should notify you when it's done—or blocked.

12. Reviewing Output and Chaining Workflows

Completion is not the end of responsibility.

12.1 Human-in-the-Loop Is Non-Negotiable

When a workflow finishes, it produces a deliverable:

- Document
- Link
- Email
- Dataset

Always review the output before proceeding.

Catching issues early prevents downstream propagation.

12.2 Chaining Without Compounding Error

Individual workflows are useful.
Chained workflows are leverage.

Example:

- Workflow 1: Lead scraping
- Workflow 2: Email enrichment
- Workflow 3: Personalized first-line generation

This modular chain **outperforms** a single monolithic workflow attempting all steps at once—without increasing error rates.

12.3 When the Agent Needs Help

Sometimes the agent will stop and ask:

- What it was trying to do

- What went wrong
- What options exist to fix it

Respond plainly. Point it in the right direction. Continue—or stop and restart with corrected assumptions.

12.4 Sensitivity-Based QA Rules

- **High sensitivity tasks** → Always QA
  Customer-facing content, financial documents, invoices, proposals
- **Low sensitivity tasks** → Can remain automatic
  Scraping, transformations, logic-tested outputs

And some things simply **should not be automated**:

- Live voice interactions
- High-stakes real-time communication

---

13. 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.

13.1 From Design to Execution

- LLMs design workflows
- Execution scripts run deterministically
- Scripts only need a trigger

Just say:

> "Turn this into a cloud function."

Example:

- Deploy Python functions via Modal using a webhook URL

13.2 What a Webhook Really Is

A webhook is simply:

- A URL
- Receiving an HTTP POST
- Triggered by an external event

Events can include:

- Purchases
- Commits
- Comments
- Scheduled jobs

13.3 When to Deploy to the Cloud

Deploy when workflows are:

- Scheduled (daily reports, weekly summaries)
- Recurring
- Event-driven

13.4 Observability Is Mandatory

Cloud failures are harder to diagnose than local ones.

Requirements:

- Explicit logging
- Centralized status reporting
- Failure notifications

Example:

- Slack channel `#agentic-cloud-log`
- Every workflow posts success or failure
- Failures link directly to deployment logs

Prompt example:

> "Give me a status check on all my Modal deployments."

13.5. Scheduled Deployment

Scheduling is just automation with accountability.

Example directive:

> "Deploy this workflow on a cron schedule."

Options:

- Every 5 minutes
- Mondays, Wednesdays, Fridays at 9 AM
- Daily at midnight

14. Scaling via Context Isolation

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

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

14.1 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.

Now that you get the basics, let's build on it...

15. Sub-Agents

As agentic systems scale, their most common failure mode is not logic—it's **context pollution**.

15.1 The Context Pollution Problem

Context pollution occurs when intermediate reasoning, partial attempts, retries, and exploratory work accumulate in a single conversation window. Over time, this clutter degrades performance.

There is a well-documented relationship between **context length and error rates**:

- More tokens → more opportunities for mistakes
- All else equal, longer contexts are noisier and less reliable

This is not theoretical. It is observable behavior across models and workloads.

15.2 How Sub-Agents Solve the Problem

A sub-agent is a deliberately isolated, task-scoped AI worker that operates in its own fresh context window, performs a specific job, and returns only its relevant outputs to a parent (orchestrating) agent.

In other words: a sub-agent does the messy work elsewhere so the main agent stays clean, focused, and reliable.

Sub-agents address context pollution through **isolation**.

Instead of forcing one agent to:

- Think
- Explore
- Fail
- Recover
- And still stay sharp

You delegate the messy work elsewhere.

Each sub-agent:

- Operates in its **own fresh context window**
- Performs a narrowly scoped task
- Returns only **relevant, distilled results** to the parent agent

This mirrors how modern models already behave internally:

- Intermediate “thinking” is discarded
- Only the final output is retained
- Token budgets stay lean

Sub-agents externalize that same principle.

15.3 Performance Evidence

Empirically, this approach works.

Data shows that:

- **Opus-based lead agents using Sonnet sub-agents outperform single-agent Opus setups by ·90% on research-heavy tasks**

The gains come not from smarter reasoning—but from **cleaner context**.

15.4 The Tradeoffs

Sub-agents are not free.

There are real costs:

- Increased implementation complexity
- More moving parts
- Risk of error compounding across agent boundaries

This is why restraint matters.

Recommendation:

> Stick to **one or two sub-agent types** initially. Do not over-orchestrate.

Here's two I like:

- Reviewer subagent - the Orchestrator wrote the code so biased towards its "correctness - like a writer proofreading their own work immediately after writing it

> Example prompt: "I'd like to create a subagent. The idea behind the subagent is it will look at the execution scripts another agent develops with fresh eyes and determine if it is done effectively or efficiently. It will then provide instructions to the top-level agent which can then apply the guidance to improve the quality of the build"...

"After you create any script, use the reviewer subagent to check for its quality"

- Documenter subagent - Updates your directives based on what the system has learned over time

Ensures self-annealing executions get documented; read access to all files, write access to directives

15.5 High-Leverage Sub-Agent Patterns

Two sub-agent types deliver outsized returns with minimal complexity.

REVIEWER SUB-AGENT

The orchestrating agent is biased toward its own output—like a writer proofreading immediately after writing.

A reviewer sub-agent provides:

- Fresh eyes
- Independent judgment
- Reduced confirmation bias

Use

16. Final Thoughts

The future is cloud-native agents.

Imagine:

- Sending a natural-language query to a URL
- `basepath/?action=do_the_thing`
- A reliable, observable workflow executes

Just systems, designed correctly.

16.1 Next Steps

You have two paths forward:

1. **Self-Serve** — Download `AGENTS.md` from my downloads link https://collinwilkins.com/agents, insert 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.

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"
```