Executable Book

MCP with DSPy Theory and Application

Building Spec-Driven, Policy-Enforced Agents with MCP, DSPy, and PKM

Jason T. Cole

First working edition

An executable book whose claims are reinforced by specifications, code, tests, and derived docs.

learning-mcp-agent

Published via learning-mcp-agent.

Project repository: https://github.com/jtcole/learning-mcp-agent

Edition: First working edition.

Contents

Chapter 00 — Book Contract

0.1 Book Contract — MCP with DSPy Theory and Application

This cluster defines the **structural and procedural contract** for the book.

It specifies: - what constitutes a chapter, - required invariants for promotion, - lifecycle stages from Zettel → Chapter → Skill, - enforcement modes (local vs GitHub), - and failure modes of large, evolving knowledge bases.

No domain content belongs here.
This contract governs *all other chapters*.

Contract Sections

- [[What_Constitutes_a_Chapter]]
- [[Required_Chapter_Invariants]]
- [[Promotion_Lifecycle_of_Knowledge]]
- [[Failure_Modes_of_Large_Knowledge_Bases]]

Acceptance Criteria

- This MoC exists as the **only** book-level contract.
- All chapter MoCs comply with the rules defined here.
- Promotion tooling enforces these rules in GitHub-mode.

0.2 What Constitutes a Chapter

A **chapter** is a *coherent, promotable unit of knowledge*.

A chapter is not prose length or topic count — it is a **contractual unit**.

A Chapter MUST Have

- Exactly **one MoC** (`MoC_Chapter_<N>_<Title>.md`)
- A clearly defined **intent**
- An ordered list of **atomic Zettels**
- Explicit **Acceptance Criteria**
- Zero ambiguity about scope

A Chapter MAY Have

- A generated walkthrough under `docs/book/`
- Executable tooling references
- Promotion-to-skill mappings

A Chapter MUST NOT

- Span multiple MoCs
- Depend on unstated conventions
- Require manual interpretation to promote
- Mutate vault state implicitly

Implication

If a cluster cannot be promoted deterministically,
it is not yet a chapter.

0.3 Required Chapter Invariants

Every chapter in this book MUST satisfy the following invariants.

These are enforced mechanically where possible.

Structural Invariants

- One and only one MoC per chapter
- All atomic notes live under the chapter directory

- All wikilinks are vault-relative
- Filenames are deterministic and stable

Semantic Invariants

- Chapter intent is explicit
- Atomic notes are conceptually independent
- No duplicated conceptual authority
- Failure modes are acknowledged

Promotion Invariants

- Acceptance Criteria are present in the MoC
- Promotion is idempotent
- GitHub-mode blocks non-conforming chapters
- Local mode may warn but must not silently fix

Walkthrough Invariants

- Generated walkthroughs introduce **no new ideas**
- Walkthroughs are derived artifacts
- Canonical content lives only in the vault

Breaking an invariant blocks promotion.

0.4 Promotion Lifecycle of Knowledge

Knowledge in this system progresses through explicit stages.

There are no implicit promotions.

Lifecycle Stages

1. **Atomic Zettel**
   - Single idea
   - No authority
2. **Chapter Cluster**

- Structured aggregation
- Governed by a MoC

3. **Promoted Chapter**
   - Meets acceptance criteria
   - Regenerated into book walkthrough
4. **Skill**
   - Executable contract
   - CI-enforced behavior

Promotion Properties

- Promotions are explicit, logged, and reversible
- Promotion tooling never invents content
- Failure to promote is a signal, not an error

Key Rule

> *If knowledge cannot survive promotion, it is not ready to be trusted.*

0.5 Failure Modes of Large Knowledge Bases

This book is designed to resist known failure modes.

Common Failure Modes

- Orphaned notes with no authority
- Multiple MoCs claiming the same scope
- Implicit conventions known only to the author
- Silent drift between prose and tooling
- Generated artifacts treated as canonical

Structural Defenses

- One-MoC rule
- Acceptance criteria enforcement

- GitHub-mode strict execution
- Promotion as a gate, not a copy

Cultural Failure Mode

Treating agents or tools as peers instead of apprentices.

This book explicitly rejects that model.

Guiding Principle

> *Structure exists to preserve meaning under scale.*

1

Chapter 01: Foundations

1.1 The Problem Space

Modern computational workflows increasingly suffer from fragmentation: tools do not speak to one another, data lives in silos, and reasoning happens in disconnected layers. As systems grow in complexity, the cognitive overhead on the human grows instead of shrinking.

Agentic automation emerges as a necessary response.
Instead of scripting every interaction explicitly, we allow structured agents to reason, plan, and execute using standardized interfaces and external tools.

The problem space is defined by:

- proliferating digital tools
- inconsistent interfaces
- rising cognitive load
- brittle automation scripts
- lack of reusable abstractions
- disconnected human vs machine knowledge stores

This motivates the convergence of MCP, Spec-Driven Development, DSPy, and PKM.

1.2 What is MCP

The Model Context Protocol (MCP) defines a universal way for LLMs to interact with external tools, resources, and data systems. It transforms tools into discoverable, typed API-like interfaces that the model can reason about and invoke autonomously.

Key ideas:

- Tools as protocol-described capabilities
- Strong typing via JSON schemas
- Context and state managed across tool calls
- Portable across agents, models, and environments

MCP solves the fragmentation problem by standardizing how software exposes functionality to agents, enabling interoperability and composability at scale.

1.3 Spec Driven Development

Spec-Driven Development (SDD) shifts the engineering focus from code-first to interface-first design. Instead of writing code and documenting it afterward, SDD requires that developers write structured specifications defining:

- behaviors
- inputs and outputs
- constraints
- variants
- testable interfaces

The implementations are then generated, validated, or scaffolded by tools like `specify` and `codex`.

This approach:

- reduces drift
- improves clarity
- accelerates onboarding
- increases testability
- aligns perfectly with MCP tool definitions

SDD is the natural complement to MCP, offering a disciplined way to define agent tool behavior.

1.4 Why DSPy Matters

DSPy provides a systematic method for converting declarative agent logic into optimized LLM programs. Unlike prompt engineering, DSPy compiles agent behavior through:

- Signatures
- Modules
- Optimizers
- Compositional reasoning structures

It is the missing optimization layer for agent tool use.
Where MCP defines *what* tools exist and *how* they behave, DSPy defines *how the agent should think* when using those tools.

DSPy improves:

- reliability
- reasoning depth
- tool selection accuracy
- multi-step execution

DSPy is essential for scaling agent systems from handcrafted prototypes to robust, production-level behavior.

1.5 PKM and Agents

Personal Knowledge Management (PKM) systems like Obsidian are knowledge substrates, not just note containers. They form the long-term memory layer upon which agentic systems can reason and act.

Agents need:

- structured conceptual maps
- durable knowledge graphs
- semantic links
- domain ontologies
- evolving reference material

Zettelkasten provides this through:

- atomicity
- bidirectional linking
- emergent structure via MoCs
- reusability of concepts

Agents + PKM create a hybrid intelligence loop:
your vault becomes the agent's brain, and the agent becomes your cognitive amplifier.

1.6 Foundations Summary

The foundational concepts underpinning this book converge on a single insight:
modern agentic systems require rigorously defined tool interfaces (MCP), interface-first engineering (Spec-Driven Development), systematic optimiza-

tion of reasoning processes (DSPy), and a durable human knowledge substrate (PKM).

When combined, these create a unified architecture where:

- MCP exposes capabilities
- SDD formalizes behavior
- DSPy optimizes cognition
- PKM organizes long-term knowledge
- Agents execute across them

This chapter frames the vocabulary and mental models needed for the deeper engineering work ahead.

2

Chapter 02: Constitution Layer

2.1 Purpose of Agent Constitution

An **agent constitution** is a meta-level structure that governs how an AI agent behaves, reasons, and interacts with tools. It defines the *rules of agency*: what the system must do, may do, must not do, and how it should resolve ambiguity.

Without a constitution, agents drift. Their behavior becomes inconsistent, fragile, or inefficient. A constitution acts as:

- a **behavioral contract**
- a **set of operating principles**
- a **decision framework**
- a **mechanism for generalization across tasks**
- a **binding layer across specs, tools, and model reasoning**

For multi-tool MCP systems, the constitution prevents chaos by establishing stable norms the agent follows regardless of context.

2.2 Global Principles and Guarantees

Global principles define the **stable behavioral invariants** an agent must uphold. These principles do not change between tasks, domains, or toolsets.

Examples of global principles:

- **Clarity:** The agent must reason explicitly and avoid ambiguous interpretations.

- **Honesty:** The agent must avoid fabrication and signal uncertainty precisely.
- **Assistance Orientation:** The agent prioritizes user goals while respecting constraints.
- **Reversibility:** Actions should be chosen to minimize irreversible outcomes unless explicitly permitted.
- **Transparency:** The agent should internalize and expose its reasoning chain when beneficial.

These principles establish the *character* of the agent.
They are higher-level than tool contracts and independent of implementation.

2.3 1. Hard Constraints

Safety rules constrain **how** an agent may act, regardless of its capabilities. They form the *boundary layer* for all decisions.

Categories of constraints:

1. Hard Constraints These are non-negotiable: - Do not execute destructive or irreversible tool calls without explicit user approval. - Do not exceed specified tool limits. - Do not fabricate tool outputs.

2. Soft Constraints These guide good behavior but allow flexibility: - Prefer interpretable steps. - Choose the safest viable strategy when uncertain.

3. Compliance Rules These map to legal, organizational, or ethical requirements such as: - Data retention policies
- Privacy restrictions
- Corporate governance restrictions
- MCP-level tool access permissions

Together, these create a **safety envelope** for the agent's reasoning.

2.4 Naming Conventions

Naming and versioning rules ensure that an agent's tools, signatures, and reasoning modules remain navigable and stable.

Naming Conventions

- Use descriptive, human-readable identifiers.
- Prefer nouns for tools, verbs for signatures, adjectives for variants.
- Avoid duplication or synonym collisions.

Versioning Conventions

- Use semantic versioning for tools and specs.
- Minor versions introduce non-breaking improvements.
- Major versions indicate conceptual or structural shifts.

Structural Conventions

- Constitutions reference stable IDs rather than filenames.
- Related rules must remain atomic to prevent cascading drift.
- Constitutions evolve linearly and explicitly—never implicitly.

These conventions ensure durability, reproducibility, and scalability across agent generations.

2.5 Constitution → Specs

The constitution is the **governing layer**, but it does not replace specs, MCP interfaces, or DSPy modules. Instead, it *coordinates* them.

Constitution → Specs Specs define *what* tools do.
The constitution defines *how* the agent should behave when using them.

Constitution → MCP MCP exposes capabilities.
The constitution enforces behavioral norms when invoking those capabilities.

Constitution → DSPy DSPy optimizes reasoning patterns.
The constitution sets the normative boundaries for acceptable reasoning.

Together, these create a coherent agent architecture where:

- Specs formalize interfaces
- MCP exposes tools
- DSPy optimizes cognition
- The constitution governs behavior

This unifies the agent, ensuring stability across tasks and environments.

3

Chapter 03: Spec-Driven Development Workflow

3.1 What is a Spec

A **specification** (spec) is a structured description of how a software component should behave.
It defines *interfaces*, *constraints*, *inputs*, *outputs*, and *allowed behaviors*—not implementation details.

Specs are not documentation.
They are **executable contracts** between human intent, tools, and agents.

A high-quality spec ensures:

- consistent expectations across collaborators
- clarity before implementation
- testable behavior
- a stable surface for tools like MCP and DSPy

3.2 Anatomy of a Spec

A high-quality spec contains several core components:

- **Behavioral Description** — What the system must/may do.

- **Inputs/Outputs** — Typed, validated, explicit.
- **Constraints** — Rules that restrict behavior.
- **Variants** — Conditional behavior branches.
- **Error Model** — Expected failure conditions.

These elements allow Codex, Specify, MCP tools, and DSPy modules to reason about the system with clarity.

3.3 Spec First Workflow

Spec-First engineering shifts design from implementation-driven to interface-driven development.

Workflow:

1. **Think** — clarify conceptual behavior.
2. **Structure** — identify inputs, outputs, constraints.
3. **Write** — encode the spec using `specify`.
4. **Refine** — iterate through validation.
5. **Generate** — produce tool definitions, signatures, or tests.

This prevents architectural drift and enhances modularity.

3.4 Spec to MCP

Specs translate directly into MCP tool definitions.

Mapping:

- Spec **inputs** → MCP tool `input_schema`
- Spec **outputs** → MCP tool `output_schema`

- Spec **constraints** → validation and guards
- Spec **errors** → structured MCP error signatures

This enables consistent, version-controlled tool behavior exposed to agents.

3.5 Spec to DSPy

Every spec produces a **DSPy signature**, defining:

- expected inputs
- required reasoning targets
- output transformations
- constraints on behavior
- optimization structure

DSPy compiles these signatures into optimized LLM reasoning modules.

3.6 SDD Feedback Loop

The SDD feedback loop ensures that specs, MCP tools, and DSPy modules evolve coherently.

Loop steps:

1. **Define** — write/extend the spec
2. **Generate** — tools, signatures, tests
3. **Execute** — run tools / agents
4. **Observe** — collect failures & drift
5. **Refine** — update the spec; repeat

This creates a virtuous cycle of clarity → execution → refinement.

4

Chapter 04: MCP Server Fundamentals

4.1 What is an MCP Server

An **MCP server** exposes tools—structured, validated functions—that an AI agent can call.
The server is not an LLM; it is a capability layer.

Key roles:

- Host capabilities the agent can invoke
- Validate inputs through schemas
- Produce structured outputs
- Provide deterministic behaviors
- Serve as an execution sandbox for computational work

The agent becomes the reasoning layer; the server becomes the action layer.

4.2 MCP Protocol and Sessions

The **MCP protocol** defines how clients and servers communicate.

Core elements:

- **Request/Response Messages** — The client issues a tool call; the server responds.
- **Session State** — Lightweight context maintained across calls.
- **Lifecycle** — Discover → Invoke → Respond → Resolve.
- **Transport-Agnostic Design** — MCP works over any channel (stdio, WebSocket, etc.).

The protocol enforces structure so agents have predictable interaction patterns.

4.3 Tool Definitions and Schemas

Every MCP tool includes:

- **name** — unique identifier
- **description** — human-readable summary
- **input_schema** — JSON Schema validating inputs
- **output_schema** — JSON Schema validating outputs
- **errors** — structured failure signatures

Schemas enforce correctness and stability.
Ideally, they should be mechanically generated from **specify** specs.

4.4 MCP Error Models

MCP tools expose **structured error signatures**.

Types of errors:

- **Validation Errors** — Schema mismatch
- **Execution Errors** — Tool logic failed

- **Contract Violations** — Behavior deviates from spec
- **Expected Failures** — Part of normal operation (e.g., "file not found")

Error models help agents reason safely under uncertainty.

4.5 Server Capabilities and Versioning

Capabilities describe what the server offers:

- list of tools
- metadata
- server version
- capability extensions

Versioning supports:

- safe upgrades
- deprecations
- feature negotiation
- multi-agent compatibility

Specs should evolve in lockstep with capability changes.

4.6 State and Statelessness in MCP

MCP tools should be **stateless** unless necessary.

Why?

- predictable behavior
- easier testing

- no hidden context
- reproducibility

But *sessions* may hold lightweight state such as:

- active file context
- ongoing conversation
- cached environment variables

State-heavy logic belongs **outside** the tool itself, often handled by DSPy modules.

4.7 MCP with DSPy and PKM

MCP, DSPy, and PKM form a **three-layer agentic system**:

- **MCP** → Performs actions (tools)
- **DSPy** → Performs reasoning (signatures & modules)
- **PKM** → Provides memory, documents, content, workflows

The PKM agent you're designing will:

1. Use MCP tools for interacting with Obsidian
2. Use DSPy models for planning & reasoning
3. Use your Zettelkasten as long-term semantic knowledge

This triad unlocks a modular AI system that learns, acts, and evolves.

5

Chapter 05: DSPy Framework Deep Dive

5.1 Why it matters:

DSPy is a **declarative optimization framework** for teaching LLMs how to reason through clearly defined interfaces.

Unlike prompting, DSPy uses:

- **Signatures** that declare inputs, outputs, and constraints
- **Modules** that learn behaviors
- **Optimization** that adjusts reasoning strategies

DSPy treats reasoning steps as *trainable programs*, not static prompts.

Why it matters:

- predictable behavior
- modular reasoning
- tool integration
- controlled autonomy

5.2 DSPy Signatures

A **DSPy signature** defines the structure of a reasoning task.

Components include:

- **Inputs** — Required context
- **Outputs** — Expected structured result
- **Constraints** — Logical or semantic rules
- **Behaviors** — What reasoning must accomplish

Signatures resemble SDD specifications and can be derived from specs directly.

5.3 DSPy Modules

DSPy **modules** are learned reasoning components that fulfill signatures.

Types include:

- **Rewrite Modules** — Transform text or structures
- **Planning Modules** — Break tasks into steps
- **Retrieval Modules** — Pull relevant context
- **Agentic Modules** — Decide which tools to call

Modules can be composed into reasoning pipelines, enabling complex behaviors.

5.4 Key Ideas:

DSPy uses **declarative optimization** to improve reasoning:

- No gradient descent
- No retraining of base models

- Optimization occurs at the *behavioral level*

DSPy evaluates candidate reasoning paths against scoring functions and rewrites the internal reasoning program.

Key Ideas:

- evaluation functions
- constrained optimization
- self-improving reasoning graphs

5.5 DSPy orchestrating MCP

DSPy modules can **call MCP tools** as part of their reasoning.

DSPy learns:

- when to call a tool
- which tool to call
- why the tool is relevant
- how tool outputs feed into next reasoning steps

DSPy becomes the **decision layer**, MCP becomes the **action layer**.

5.6 DSPy Memory and PKM

DSPy supports multiple memory types:

- **External memory** (PKM vault, structured documents)
- **Retrieval-based memory**
- **Module-learned memory** via optimization

Your PKM system becomes a **semantic backbone**, enabling:

- long-term recall
- structured reference retrieval
- context-aware reasoning

5.7 DSPy Reasoning Graphs

DSPy composes signatures and modules into **reasoning graphs**.

A reasoning graph defines:

- a sequence of reasoning steps
- data flow between modules
- tool-calling opportunities
- branches and fallback strategies

DSPy optimizes the *entire graph*, not individual calls.

5.8 DSPy as PKM Agent

DSPy becomes the **reasoning engine** of your personal PKM agent.

The agent integrates:

- **MCP** → action capabilities
- **DSPy** → planning & reasoning
- **PKM** → long-term knowledge

Benefits:

- contextual awareness
- structured memory
- safe, explainable reasoning

- modular extensibility

6

Chapter 06: PKM Integration with Obsidian

6.1 PKM in Agent Systems

PKM provides **long-term semantic memory** for agent systems.
Unlike ephemeral model context, PKM persists over time and becomes the agent's stable source of truth.

PKM allows: - durable knowledge accumulation
- predictable retrieval
- explainable reasoning
- memory across sessions

Zettelkasten introduces *atomicity*, *linking*, and *evergreen structure*, aligning naturally with agentic reasoning.

6.2 Zettelkasten as Knowledge Graph

Zettelkasten is a **machine-readable conceptual graph**:

- notes = nodes
- links = edges
- metadata = schema
- tags = semantic clustering

Because each note is atomic and explicitly linked, agents can traverse the knowledge graph predictably.

6.3 Obsidian as PKM OS

Obsidian functions as an **operating system for PKM**:

- Markdown-based durability
- Plugins for automation
- Graph visualization
- Backlink support
- YAML frontmatter integration

Its structure is ideal for integration with MCP tools and DSPy retrieval.

6.4 Designing a PKM Vault

A PKM vault for agent use must be:

- stable
- predictable
- explicit in structure

Recommended folders:

- `Pure Zettel/`
- `MoC/`
- `Projects/`
- `Resources/`

Agents must reliably locate notes and metadata.

6.5 Zettel Metadata and Schema

Metadata gives structure to PKM:

- IDs
- timestamps
- tags
- sources
- revision numbers

This schema must be machine-readable so MCP tools and DSPy modules can use it.

6.6 MCP Tools for PKM

MCP enables structured PKM interaction.

Recommended tools:

- `read_note`
- `write_note`
- `search_notes`
- `list_links`
- `compute_embeddings`

Safety considerations include preventing note corruption and enforcing schema rules.

6.7 DSPy and PKM Retrieval

DSPy retrieves PKM content using:

- retrieval signatures
- evaluation functions
- embedding similarity
- metadata filtering

DSPy must decide: - when to retrieve
- what to retrieve
- how to integrate retrieved notes into reasoning

6.8 PKM as Agent Identity

PKM becomes the agent's **identity layer**.

The vault defines:

- the agent's persistent knowledge
- its worldview
- its stable concepts
- how it interprets new information

PKM should evolve but remain coherent.

6.9 PKM MCP DSPy Loop

The PKM–MCP–DSPy loop forms a **continuous learning engine**:

1. DSPy formulates queries
2. MCP retrieves structured PKM notes
3. DSPy reasons over them
4. MCP writes updates

5. PKM grows
6. Reasoning improves

This creates a self-improving agent grounded in real, stored knowledge.

7

Chapter 07: MCP + DSPy + PKM Agent Project

7.1 What is an Agent

An **agent** is a system that can:

1. Perceive inputs (questions, environment state, documents)
2. Reason about them (plan, evaluate options)
3. Act in the world (via tools or side effects)
4. Learn or adapt over time (update memory, strategies, or both)

In this book's context, an agent is *not* just:

- a single LLM call
- a chat interface
- a brittle tool-calling chain

Instead, it is a layered system whose behavior emerges from:

- long-term memory (PKM)
- structured reasoning (DSPy)

- external capabilities (MCP tools)
- safety and alignment rules (Constitution)

7.2 Mermaid Diagram — Four-Layer Agent Model

This book adopts a **strict four-layer architecture** for agents:

1. **PKM Layer** — Long-term semantic memory and identity (Obsidian + Zettelkasten).
2. **DSPy Layer** — Reasoning, planning, and decision-making.
3. **MCP Layer** — Concrete actions and capabilities (tools and servers).
4. **Constitution Layer** — Safety rules, constraints, and behavioral guarantees.

These layers are *conceptually distinct* and communicate via explicit interfaces.

Mermaid Diagram — Four-Layer Agent Model

```
flowchart TB
    subgraph Constitution_Layer [Constitution Layer]
    end

    subgraph PKM_Layer [PKM Layer]
    end

    subgraph DSPy_Layer [DSPy Layer]
    end

    subgraph MCP_Layer [MCP Layer]
    end

    PKM_Layer --> DSPy_Layer
    DSPy_Layer --> MCP_Layer
    MCP_Layer --> DSPy_Layer
    DSPy_Layer --> PKM_Layer

    Constitution_Layer --- PKM_Layer
```

```
Constitution_Layer --- DSPy_Layer
Constitution_Layer --- MCP_Layer
```

The Constitution constrains all three operational layers, but is kept conceptually distinct so it can be versioned, audited, and reasoned about independently.

7.3 PKM Layer

The **PKM Layer** is the agent's long-term semantic memory and identity.

It contains:

- atomic notes (Zettels)
- their links and structure
- metadata (sources, tags, timestamps)
- project scaffolds and maps of content

The agent uses PKM to:

- recall prior knowledge
- ground new reasoning in established facts
- accumulate learning across sessions
- explain why it believes something

The PKM layer is read and written via MCP tools and interpreted via DSPy reasoning modules.

7.4 DSPy Layer

The **DSPy Layer** is responsible for *reasoning and planning*.

It operates over:

- input queries

- retrieved PKM notes
- available MCP tools
- constitutional constraints

DSPy expresses reasoning as:

- **signatures** (what needs to be computed)
- **modules** (how reasoning is carried out)
- **optimization** (improving behaviors over time)

In this architecture, the DSPy layer decides:

- when to call tools
- what context to retrieve from PKM
- how to sequence multi-step reasoning graphs

7.5 MCP Layer

The **MCP Layer** is the agent's action surface.

It exposes:

- tools for file operations
- PKM read/write operations
- search and retrieval capabilities
- domain-specific operations (e.g., analytics, automation)

MCP tools are:

- described by JSON Schemas
- derived from specifications when possible

- versioned and capability-negotiated

The DSPy layer calls MCP tools; MCP never calls DSPy directly.
This preserves a clear separation between *reasoning* and *action*.

7.6 Constitution Layer

The **Constitution Layer** defines what the agent *may* and *may not* do.

It constrains:

- which tools may be called
- what kinds of PKM updates are allowed
- how sensitive information is handled
- how conflicts and ambiguities are resolved

The Constitution is:

- written in natural language plus structured rules
- versioned and auditable
- applied consistently across reasoning and action

DSPy modules and MCP tools must both respect constitutional constraints.

7.7 Mermaid Diagram — High-Level Agent Loop

An agent run can be described as a **lifecycle**:

1. **Perception** — Receive a query or environment signal.
2. **Context Building** — Retrieve relevant PKM notes and metadata.
3. **Planning** — DSPy constructs or selects a reasoning graph.
4. **Action** — MCP tools are invoked as needed.

5. **Evaluation** — Results are checked against goals and constraints.

6. **Memory Update** — PKM is updated with new insights if appropriate.

Mermaid Diagram — High-Level Agent Loop

```
flowchart LR
    UserQuery[User / Environment Input]
    Context[PKM Context Retrieval]
    Plan[DSPy Planning]
    Act[MCP Tool Calls]
    Evaluate[Evaluate & Check Constitution]
    Update[Update PKM (Optional)]
    Reply[Return Answer]

    UserQuery --> Context --> Plan --> Act --> Evaluate --> Reply
    Evaluate --> Update
    Update --> Context
```

This loop is executed under the watch of the Constitution Layer, which can block or reshape plans and actions.

7.8 Mermaid Diagram — Simplified Reasoning Graph

A **reasoning graph** for a PKM agent is a structured set of DSPy signatures and modules.

Example structure:

- Node 1: Interpret user intent
- Node 2: Retrieve relevant PKM notes
- Node 3: Synthesize an answer draft
- Node 4: Decide whether to call tools
- Node 5: Post-process answer and suggest PKM updates

Mermaid Diagram — Simplified Reasoning Graph

```
flowchart TD
    N1[Interpret Intent]
    N2[Retrieve PKM Context]
    N3[Synthesize Draft Answer]
    N4[Decide on Tool Use]
    N5[Refine & Propose Updates]

    N1 --> N2 --> N3 --> N4 --> N5
```

DSPy treats each node as a trainable, optimizable unit, enabling the agent to improve its behavior over time.

7.9 Agent Failure Modes and Recovery

Agents can fail in several ways:

- **Retrieval failure** — PKM returns nothing useful or conflicting notes.
- **Tool failure** — MCP tools error out or violate contracts.
- **Reasoning failure** — DSPy selects a poor reasoning path.
- **Constitutional conflict** — Planned actions violate safety rules.

Recovery strategies include:

- fallback reasoning graphs
- alternative retrieval strategies
- degraded but safe behaviors
- explicit deferral to the human user

Robust agents are designed with explicit failure-handling pathways rather than relying on best-case behavior.

7.10 Mermaid Diagram — Example Query Cycle

Consider a user query:

> *"Summarize my recent thinking about MCP servers and suggest the next three steps to implement my PKM agent."*

A single agent run might proceed as:

1. **Perception (Query Ingest)**
   - The DSPy layer receives the query and classifies it as a *PKM-aware planning request*.
2. **Context Building (PKM Layer)**
   - MCP tools search Zettels tagged with `mcp`, `agents`, and `pkm`.
   - Relevant notes from Chapters 4–6 are retrieved.
3. **Planning (DSPy Layer)**
   - A reasoning graph is selected that:
     - interprets the goal,
     - structures prior knowledge,
     - produces a set of actionable steps.
4. **Optional Actions (MCP Layer)**
   - The agent might call tools to inspect the current project folders, specs, or code repositories.
5. **Evaluation and Constitution Check**
   - The proposed steps are checked against constitutional rules (e.g., no destructive file actions).
6. **Memory Update (PKM Layer)**
   - A new Zettel is drafted (or suggested) summarizing the plan.
   - The user may accept or edit before it is written.
7. **Response to User**
   - The agent returns:
     - a summary of current MCP-related knowledge
     - three concrete next steps
     - a note that a new planning Zettel has been proposed.

Mermaid Diagram — Example Query Cycle

```
sequenceDiagram
    participant U as User
    participant D as DSPy Layer
    participant P as PKM Layer
    participant M as MCP Layer
    participant C as Constitution

    U->>D: Query: MCP servers + next steps
    D->>P: Retrieve MCP + PKM notes
    P-->>D: Relevant Zettels
    D->>D: Plan reasoning graph
    D->>M: Optional tool calls (inspect project state)
    M-->>D: Tool results
    D->>C: Check plan & actions
    C-->>D: Approve / adjust
    D->>P: Propose new planning Zettel
    P-->>D: Confirm write (or user approval)
    D-->>U: Answer + next steps + note update
```

This example illustrates how all four layers participate in a single coherent agent run.

8

Full-Stack Example — `write_note`

8.1 Full-Stack `write_note`: Why this is the first example

`write_note` is the smallest end-to-end slice that touches all layers:

- **Spec** (tools.yml)
- **Validation** (schema + arg checks)
- **Implementation** (server routes to vault write)
- **Safety policy** (no silent overwrite)
- **Tests** (integration tests enforce behavior)
- **Example** (temp-vault walkthrough script)

Starting with an existing tool forces the loop to prove *process quality* instead of "feature velocity".

8.2 Full-Stack `write_note`: Overwrite semantics

The system forbids **silent overwrites**:

- A second write that maps to the same filename must **fail** unless `overwrite=true`.
- If `overwrite=true`, the response must explicitly report whether a file was overwritten (e.g., `overwritten: true/false`).

Why it matters for PKM: - Notes are "knowledge artifacts"; silent overwrites destroy trust. - Agents must be constrained to require explicit intent for mutation.

8.3 Full-Stack write_note: Specs, tests, and examples form a contract

A spec-only system drifts. A code-only system is opaque.

The stable triangle is:

- **Spec** declares the tool contract (inputs/outputs).
- **Tests** enforce the contract and safety invariants.
- **Examples** demonstrate correct usage in a reproducible way (temp vault by default).

If any corner changes, the other two must change too. Otherwise: the "tool" becomes a rumor.

8.4 Full-Stack write_note: From Zettels → Skills → Code

- **Zettels** capture intent ("no silent overwrites", "tests enforce philosophy").
- **Skills** operationalize intent as a repeatable procedure with STOP conditions.
- **Code + tests** are the enforcement mechanism.

Skills should **follow** Zettels: the procedure is derived from already-reasoned design.

Related cluster: - [[MoC_From_Zettels_to_Skills]]

8.5 Full-Stack write_note: What to copy for the next tool

When adding the next tool, copy this pattern:

1. Update the spec (tool contract first).
2. Implement routing in the server with validation.
3. Add tests for:
   - happy path
   - invalid args
   - safety policy (if any write/mutation exists)

4. Add a walkthrough example that defaults to temp vault.
5. Update the book MoC to link the new MoC + walkthrough.

The compounding phase works when "add tool" becomes **mechanical**.

9

Full-Stack Example: search_notes (MoC)

9.1 search_notes is the canonical read-only discovery tool

`search_notes` is the primary way an agent scans the vault without risking mutation. It indexes only markdown notes with frontmatter, then surfaces ids, titles, tags, and context snippets so downstream tools can decide whether to read or ignore a candidate note.

The tool keeps parity between how humans browse the vault and how agents triage it: it honors titles and tags, finds matches in both metadata and body, and returns the filesystem path so callers can confirm provenance. Because it short-circuits at the requested limit and does not change any files, it is the safe default for discovery in compounding loops.

9.2 search_notes contract (inputs, outputs, semantics)

Inputs - `query` (required): non-empty string; search is case-insensitive. - `limit` (optional): integer 1..50; defaults to 10 and short-circuits once satisfied. - `tags` (optional): array of strings; all listed tags must be present on a note to be included.

Semantics - Matching runs over title, tags, and body text; notes without frontmatter are skipped. - Snippet centers on the first match with ~40 chars of

lead-in and ~120 chars after; falls back to the first 160 chars or the title if needed. - Results preserve the traversal order of the vault and stop at `limit`.

Outputs - Array of objects with `id`, `title`, `snippet`, `path` (absolute), and `tags`. - No side effects; responses are pure views of current vault state.

9.3 Tests and examples enforce the search_notes contract

Integration tests gate the behavior: - `test_search_notes_returns_matches` seeds two notes and asserts the matching note id, snippet inclusion of the query, and a real path. - `test_search_notes_filters_by_tags` proves tag filters are conjunctive: only notes containing all requested tags survive. - `test_search_notes_rejects_missing_query` enforces JSON Schema validation by raising ValueError when `query` is absent.

The walkthrough script `examples/walkthrough/02_search_notes.py` defaults to a temporary vault, seeds demo notes, calls `search_notes` with a limit of 5, and prints the returned metadata. Together, the tests and example form an executable contract for agents and humans.

9.4 search_notes is safe and agent-friendly (no writes)

`search_notes` only reads files. In `PKMServer.call_tool` it routes to `vault.search_zettels` which walks the vault, reads frontmatter and bodies, and assembles view-only dictionaries. No branches mutate disk.

JSON Schema validation and tag-filter checks fail fast on bad inputs, keeping agents deterministic. The hard limit cap (50) prevents unbounded traversal in constrained sessions. Because it never touches `write_note`, the safeguards D004/D005 remain untouched while agents still get the context they need to decide next actions.

9.5 How search_notes maps to Skills and the repository constitution

Contract first: the tool spec in `modules/pkm_tools/tools.yml` defines inputs/outputs; `PKMServer.list_tools` enforces completeness (D001) and

`validate_tool_args` enforces the JSON Schema boundary (D003). Tests under `tests/integration` keep the contract executable (D008).

The new Skill in `.codex/skills/full-stack-search-notes/SKILL.md` operationalizes the read-only loop: start with `search_notes`, decide on follow-on reads, and stop before any writes. This keeps reasoning (DSPy), contract (spec/tests), and execution (MCP server) separated per D007 while honoring vault-first authorship (D009/D010).

10

Full-Stack Example — `update_note`

10.1 Path-first `update_note` safety model

- The first mutation tool is **path-first**: callers must name the exact markdown path; no filename derivation or implicit routing.
- Writes are allowed only inside `Book Zettel/MCP with DSPy Theory and Application/` or `_agent_inbox/`; anything else is rejected.
- Existing notes must contain frontmatter so ids, tags, and related_notes stay stable unless explicitly replaced.
- Mutation requires `overwrite=true`; otherwise the call fails, preventing silent updates.
- The response returns path, id, title, tags, and `overwritten=true` for traceability and downstream routing.

10.2 Consent and allowed roots for `update_note`

- Update calls are **consentful**: `overwrite` must be explicitly true; missing or false results in refusal.
- Paths are validated against the vault root and must stay inside `Book Zettel/MCP with DSPy Theory and Application/` or `_agent_inbox/`.
- Attempts to update non-existent files fail loudly (FileNotFoundError) to avoid accidental creation.
- Frontmatter is preserved by default; callers may update title/tags/related_notes/source explicitly.
- These rules operationalize D004–D006 and D011 for the mutation surface.

11

Full-Stack Example — `id_index_stat`

11.1 Why `id_index_stats` exists: observability + determinism

`id_index_stats` is a **read-only observability hook** over the vault id index. It answers: *is the id space clean enough to allow id-based operations to be deterministic?*

- Without visibility into duplicate ids, id-based mutations risk writing to the wrong note or failing late.
- The tool surfaces how many notes are indexed, how many unique ids exist, and how long index construction took, so operators can reason about scale and performance budgets.
- Surfacing duplicates proactively keeps D002 (fail early/deterministically) aligned with D004/D005 safety expectations.

11.2 D012 enforced: vault-relative paths in `id_index_stats`

`id_index_stats` must return **vault-relative POSIX paths** for any duplicate evidence. Absolute paths leak machine-local details and break portability; relative paths keep the contract stable across environments.

- D012 requires agent-facing paths to be vault-relative. The tool includes the exact scope used and lists duplicate file paths relative to `vault_root`.
- Relative POSIX paths also prevent Windows/Posix separator drift and make comparisons deterministic in tests and logs.
- This keeps downstream agents from caching absolute paths that would later fail validation when the vault root changes.

11.3 `id_index_stats` contract and semantics

Input contract (JSON Schema): - `scope` (optional string): vault-relative scope override for indexing; defaults to allowed write roots. - `include_duplicates` (boolean, default `true`): include duplicate listings.

Output contract: - `scope`: vault-relative scope string actually used. - `total_notes`: count of distinct note paths indexed. - `unique_ids`: count of unique ids across stems and frontmatter. - `duplicates` (optional): map of duplicate ids → vault-relative POSIX paths. - `build_ms`: time to build the index (ms) for observability.

Semantics: - Read-only inspection; no writes or mutations are allowed. - Index keys include both filenames and frontmatter ids, so collisions are visible even when stems differ. - Duplicate reporting obeys D012 by returning vault-relative paths only.

11.4 How tests and examples enforce the `id_index_stats` contract

- Integration test `test_id_index_stats_reports_duplicates_with_relative_paths` seeds duplicate ids and asserts:
  - `total_notes` counts distinct note paths; `unique_ids` counts both stems and frontmatter ids.
  - `duplicates` exists and lists vault-relative POSIX paths only (no drive letters, no backslashes), locking in D012.
- Walkthrough `examples/walkthrough/05_id_index_stats.py` demonstrates read-only usage and shows JSON output, doubling as an executable sample for contract expectations.
- Together they keep the tool aligned with D002 (fail early/deterministically) and D009/D010 (contract comes from Zettels and tests, not ad-hoc docs).

11.5 How `id_index_stats` enables safe id-based mutation later

`id_index_stats` is the gatekeeper for any future **id-based mutation** tool (e.g., update-by-id):

- It exposes duplicate ids before mutation surfaces are enabled, preventing ambiguous writes and keeping the system in a "safe to mutate" state.
- By returning vault-relative evidence, it gives operators the exact files to clean up so the id space becomes deterministic.
- The same index builder is reused by `resolve_note_id`; keeping stats separate and read-only avoids side effects while still validating the index health.
- Once the index is clean, mutation tools can reuse the index logic with confidence that D011 (write fence) and D012 (path form) stay intact.

12

Full-Stack Example — append_to_no

12.1 Full-Stack append_to_note: Why append is safer than overwrites

- Append preserves existing note context and chronology; overwrites discard history and require a full rewrite of frontmatter and body.
- The agent only appends when a caller asserts intent (`overwrite=true`), making mutation deliberate even though the payload is additive.
- Append keeps PKM trust high: readers can audit incremental additions instead of diffing entire notes.
- For journaling workloads, appending matches the mental model of "log the next event" without risking old content.

12.2 Full-Stack append_to_note: Consent semantics + D011 write fence

- The tool refuses to run unless the caller sets `overwrite=true`, even though the action is additive; this is explicit consent for mutation.
- Writes are restricted to `Book Zettel/MCP with DSPy Theory and Application/` or `_agent_inbox/`; `_ensure_write_allowed` enforces the fence and raises `PermissionError` otherwise.
- ID-based addressing is gated behind `PKM_ENABLE_ID_INDEX=1` to avoid implicit note selection; otherwise path-only writes are allowed.
- Any attempt to escape the vault root is rejected early (`Path escapes vault root`).

12.3 Full-Stack append_to_note: D012 relative paths in mutation outputs

- After appending, the server returns `path` relative to `vault_root` (`_relative_to_root_posix`), satisfying D012.
- Vault-relative paths travel safely between machines and terminals, avoiding absolute path leakage.
- Agent callers should persist and display vault-relative paths so downstream steps (e.g., journaling viewers) remain portable.

12.4 Full-Stack append_to_note: Contract + edge cases

- Locator: exactly one of `path` or `id` is required; providing both or neither raises `ValueError`.
- File shape: target must be an existing markdown file with frontmatter; the tool refuses to create new files or append to non-markdown.
- Separator handling: `separator` defaults to double newline; the appender normalizes trailing newlines so the appended payload does not double-insert or strip blank lines.
- Frontmatter remains untouched; append operates after existing content and reports `bytes_added` for observability.
- ID resolution is opt-in via `PKM_ENABLE_ID_INDEX=1`; otherwise callers must pass a path inside the allowed roots.

12.5 Full-Stack append_to_note: Tests and examples enforce the contract

- Integration coverage asserts: explicit consent required (`overwrite=true`), exactly one locator, ID lookup opt-in, write fence respected, and vault-relative path return.
- Tests also verify newline behavior (`bytes_added` aligns with the appended payload) and that original content remains intact.
- Future walkthrough scripts should mirror test setup: seed allowed roots, default to path-based append, and log `bytes_added` for observability.
- Contract changes must update specs + tests together; otherwise the loop breaks (D008 + D009).

12.6 Full-Stack `append_to_note`: How this supports safe agent journaling

- Journaling and conversation logs expect monotonic growth; append preserves prior context while allowing fresh entries.
- The explicit-consent guard (`overwrite=true`) turns every journal write into an intentional act, reducing accidental spam.
- Separator normalization keeps entries readable (blank line between events by default) without manual newline bookkeeping.
- Vault-relative paths let downstream tools surface the updated journal location without leaking host-specific paths.

13

Full-Stack Example: list_notes (MoC)

13.1 list_notes is the discovery companion to search_notes

`list_notes` gives agents an inventory-first view of the vault, anchored to the default book scope so the first pass surfaces curated material without crafting a query. `search_notes` answers targeted questions; `list_notes` answers "what exists here?" with deterministic ordering and paging. Together they form a two-step discovery pattern: enumerate scoped, tagged assets via `list_notes`, then pivot to `search_notes` for deeper retrieval. Because both tools are read-only, agents can explore safely before deciding whether a mutation is needed.

13.2 list_notes contract (inputs, outputs, determinism)

Inputs - `scope` (optional): vault-relative root; defaults to Book Zettel/MCP with DSPy Theory and Application. - `query` (optional): case-insensitive substring against title, filename stem, and a 200-char preview. - `tags` (optional): array; all requested tags must be present on the note. - `limit` (optional): 1..500, default 50; applied after sorting. - `offset` (optional): >=0, default 0; applied after sorting. - `sort` (optional): one of `path` (default, stable lexicographic POSIX), `title` (lowercased), or `mtime` (ISO string).

Outputs - `notes`: array of objects `{path, title, id|null, mtime, tags}`; `path` is vault-relative POSIX per D012; `title` prefers frontmatter; `id` only if frontmatter exists. - `total`: count before paging. - `scope`: vault-relative scope actually used.

Determinism - Scope normalization rejects absolute paths or traversal, fixing the search root. - Sorting happens before paging, so `offset/limit` slices are stable across calls until files change. - Query and tag filters are pure predicates; no random sampling or pagination drift.

13.3 list_notes safety: read-only, scoped, D012-compliant paths

The tool never writes; it only reads markdown files and returns metadata. `_normalize_scope` rejects absolute paths, drive letters, and `..`, preventing traversal outside the vault. `_resolve_within_vault` guards against symlink escape, and missing scopes return empty results instead of failing. Paths in responses are rendered via `_relative_to_root_posix`, enforcing D012 vault-relative POSIX strings and stripping host-specific drives. Input guards enforce `limit` within 1..500 and non-negative `offset`, capping traversal. Because the walker ignores non-markdown files and honors required tags, the agent sees only scoped, compliant items.

13.4 Tests and examples enforce the list_notes contract

Integration coverage in `tests/integration/test_list_notes.py` asserts the contract: - Default call uses the book scope, excludes files outside it, and sorts paths lexicographically. - Paging and filtering preserve `total`, keep offsets stable after sort, and prove substring queries work. - Scope traversal like `../secret` raises ValueError; paths in responses are vault-relative POSIX with no drive prefixes.

The walkthrough `examples/walkthrough/07_list_notes.py` seeds a temp vault, runs `list_notes`, and prints the JSON response, providing an executable demo aligned with the spec.

13.5 Inventory-first listing reduces agent hallucination

`list_notes` grounds the agent in what actually exists before it speculates. The response carries real vault-relative paths, titles, ids (when present), and tags, so downstream reasoning can cite concrete artifacts instead of inventing notes. Deterministic sort and paging let the agent revisit slices without drift, keeping deliberation reproducible. Using the default book scope biases exploration toward curated material while still allowing scoped overrides. Coupled with `search_notes`, the agent can pivot from inventory to relevance without hallucinating structure or filenames.

13.6 Listing as a foundation for navigation patterns

Because results include `path`, `title`, `id`, `mtime`, and `tags`, `list_notes` can feed MoCs, tag indexes, and timeline views without another crawl. POSIX vault-relative paths drop directly into wikilinks or JSON outputs used by downstream compilers. Stable sort options (`path`, `title`, `mtime`) let agents build reproducible navigation structures. Tag-conjunctive filtering is already MoC-friendly: a MoC can request specific tags to generate focused tables of contents. As new navigation surfaces emerge, this tool supplies the canonical, scope-bounded inventory to populate them.

14

Pattern: Journaling & Agent Memory (No New Tool)

14.1 Append-Only as the Agent Memory Primitive

Append-only journaling is the safest memory substrate for agents because it: - preserves chronological trace without accidental rewrites (aligns with D004/D005). - keeps diffs auditable and reversible; every entry is an additive fact. - removes locking and race complexity common in in-place edits. - works with existing tools (`append_to_note` on top of a created note) so no new surface area is needed.

This pattern treats each journal entry as an immutable breadcrumb, enabling replay, debugging, and alignment checks over long sessions.

Links

- [[MoC_Journaling_Pattern]]

14.2 Daily Note Naming and Location

Convention for journaling within the D011 fence (`Book Zettel/MCP with DSPy Theory and Application/`): - Title format: `Journal YYYY-MM-DD` (sanitizes to `Journal_YYYY-MM-DD.md`), sortable and unambiguous. - All journal files live directly under the fence (no absolute paths; D012 keeps references vault-relative). - Reuse the same note for the day; all entries append to this file.

Rationale: - Predictable naming makes `list_notes/search_notes` cheap and targeted to a handful of candidates. - Staying inside the fence avoids accidental writes elsewhere in the vault.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Create_If_Missing_and_Append_Safely]]

14.3 Create-If-Missing and Append Safely

Operational recipe: 1) Probe for the daily note; if absent, call `write_note(title=Journal YYYY-MM-DD)` to create it. Default `overwrite=false` enforces D004/D005. 2) Add entries with `append_to_note(note_id=Journal_YYYY-MM-DD);` do not flip `overwrite=true` unless rerunning an idempotent migration. 3) Keep entries timestamped inside the body so the audit trail is visible when reading.

Why this matters: - Separates creation from mutation, reducing blast radius when a creation step is retried. - Aligns with append-only intent while still allowing explicit recovery paths when necessary.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Append_Only_Memory_Primitive]]

14.4 Retrieval Patterns for Journals

Preferred retrieval flow: - `list_notes(prefix="Journal_YYYY")` to bound the candidate set for the year. - `search_notes(query="Journal YYYY-MM-DD")` when exact day lookup is needed. - `read_note(note_id=Journal_YYYY-MM-DD)` to load the body for review or summarization.

Notes: - Tool responses return vault-relative paths (D012), preserving portability. - Avoid globbing across the whole vault; scoped queries keep latency low and respect D011.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Daily_Note_Naming_and_Location]]

14.5 Journaling Anti-Patterns to Avoid

- Overwriting daily notes instead of appending (breaks auditability and D005).
- Writing outside the D011 fence or using absolute paths (breaks portability per D012).
- Mixing scopes in one note (agent/system/human thoughts without headers) leading to ambiguity.
- Creating multiple files for the same day without a disambiguating suffix.
- Appending without timestamps, making sequence reconstruction fragile.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Create_If_Missing_and_Append_Safely]]

14.6 Preparing for append_journal_entry

The append-only loop can be wrapped later into `append_journal_entry` without altering behavior because: - Creation and appending are already separated; a wrapper can orchestrate `write_note` then `append_to_note` idempotently. - Naming is deterministic (`Journal YYYY-MM-DD`), so the wrapper can derive the target note without additional schema. - Retrieval remains unchanged; callers can still fall back to `list_notes`/`search_notes`/`read_note` if the wrapper is unavailable.

Design implication: build the wrapper as a thin orchestration layer, not a new storage primitive. Keep the append-only contract explicit in its schema and diagnostics.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Append_Only_Memory_Primitive]]

14.7 Journaling Pattern: Temp Vault Walkthrough

Goal: rehearse the append-only loop in a temp vault before touching the canonical vault.

Steps

- Start in temp-vault mode; confirm `vault_root` points to the sandbox location.
- Target the day's note named `Journal YYYY-MM-DD`; if it does not exist, create it with `write_note` (no overwrite flag set).
- Append entries via `append_to_note`, keeping timestamps in the entry body.
- Locate and review entries with `list_notes`/`search_notes`, then `read_note` the latest journal for memory refresh.

Guardrails

- Never overwrite journal files; append-only keeps the audit trail.
- Keep all paths vault-relative to honor D011/D012 and avoid leaking absolute paths when switching vault roots.

Links

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Create_If_Missing_and_Append_Safely]]
- [[Journaling_Pattern_Retrieval_Patterns]]

15

Chapter 13 — Promotion & Compilation Pipeline

15.1 From Zettels to Skills: Operationalizing Architectural Intent

This cluster explains how **conceptual knowledge** (**Zettels**) is transformed into **operational procedures** (**Codex Skills**) without losing architectural intent.

Zettels capture *why* and *what*. Skills encode *how* in a reusable, enforceable form.

Core Notes

- [[Zettels_Capture_Intent_Skills_Encode_Action]]
- [[Skills_as_Procedural_Zettels]]
- [[Repository_as_Execution_Context_for_Skills]]
- [[Why_Skills_Follow_Zettels_Not_Replace_Them]]

Book Placement

This material belongs between **Design Philosophy** and **Worked Implementations**.

Forward Pointer: Skills in Practice

In later chapters, the ideas in this cluster are instantiated as **Codex Skills** in the companion repository.

These Skills encode: - repository hygiene - MCP tool addition workflows - safety and overwrite guarantees

They do not introduce new ideas. They are executable forms of architectural intent already captured here.

Readers may return to this cluster when evaluating whether a Skill preserves or violates the system's design philosophy.

15.2 Zettels Capture Intent, Skills Encode Action

Zettels are optimized for **reasoning, reflection, and explanation**. They capture architectural intent, tradeoffs, and conceptual boundaries.

Skills, by contrast, are optimized for **execution**. They encode repeatable procedures that an agent can follow without reinterpretation.

The transition from Zettels to Skills marks the point where intent becomes operational.

Links

- [[MoC_From_Zettels_to_Skills]]

15.3 Skills as Procedural Zettels

Skills can be understood as **procedural Zettels**.

Like Zettels, they are: - small in scope - focused on a single idea - composable

Unlike Zettels, they are: - imperative - executable - constrained by acceptance criteria

Links

- [[MoC_From_Zettels_to_Skills]]

This makes Skills the natural operational counterpart to a Zettelkasten-based design system.

15.4 Why Skills Follow Zettels, Not Replace Them

Skills should never replace Zettels.

Zettels remain the source of truth for: - architectural rationale - design constraints - long-term understanding

Links

- [[MoC_From_Zettels_to_Skills]]

Skills are derived artifacts. They operationalize decisions that have already been reasoned about and recorded.

Reversing this order leads to brittle systems and opaque agent behavior.

15.5 Repository as Execution Context for Skills

Skills do not operate in isolation. They execute within the constraints of a repository.

The repository provides: - directory structure - specs and schemas - tests - decision logs

Links

- [[MoC_From_Zettels_to_Skills]]

This makes the repository the **runtime environment** for Skills, just as it is the constitution for agents.

16

Execution Context: GitHub-mode

16.1 GitHub-mode — No Vault Writes

GitHub-mode treats the vault as immutable during execution. Direct writes are forbidden because the repository functions as an execution sandbox rather than the source of truth for PKM.

Blocking writes prevents irreversible corruption, keeps side effects reviewable, and forces every change through deliberate human promotion.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[GitHub_Mode_Spool_Root_Semantics

16.2 GitHub-mode — Spool Root Semantics

Write spooling redirects every mutation attempt into a controlled spool root. The staging area uses deterministic paths so outputs stay predictable and git-reviewable without touching the canonical vault.

Spooling keeps autonomous agent work safe while preserving a clear handoff for human review and promotion.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[GitHub_Mode_Tool_Output_Invarian

16.3 GitHub-mode — Tool Output Invariants

GitHub-mode must produce identical outputs across local runs, CI, and GitHub Actions. Determinism depends on vault-relative paths, avoidance

of working-directory assumptions, and explicit execution-context resolution before any tool call.

When tools spool to the staging root, outputs remain predictable, diffable, and ready for promotion.

Why Agents Cannot Reason Over Absolute Paths

Absolute paths anchor reasoning to a specific machine. When an agent plans a sequence of operations using absolute paths, those plans become non-portable:

- **Local run**: `C:\Users\Alice\vault\Notes\example.md`
- **CI run**: `/home/runner/work/repo/vault/Notes/example.md`
- **Collaborator's machine**: `/Users/Bob/projects/vault/Notes/example.md`

The agent's plan breaks when executed in a different environment. Even if tools succeed locally, they fail in CI or when the repository is cloned elsewhere.

Vault-relative paths stabilize reasoning across contexts:

- **All environments**: `Notes/example.md` (relative to vault root)

This enables agents to construct multi-step plans that remain valid regardless of where the repository is mounted. The execution-context layer resolves the vault root at runtime, ensuring tool calls reference the correct absolute locations transparently.

Example: Local vs. GitHub-mode Reasoning

Local Mode (Unsafe):

```
Agent plan:
1. Read `/home/alice/vault/Book Zettel/Chapter_03.md`
2. Write summary to `/home/alice/vault/Summaries/Chapter_03_Summary.md`
```

This plan fails on Bob's machine and in CI.

GitHub-mode (Correct):

```
Agent plan:
1. Read `Book Zettel/Chapter_03.md` (vault-relative)
2. Spool summary to `staging/Summaries/Chapter_03_Summary.md` (spool-relativ
```

This plan succeeds everywhere. The execution context resolves the vault root and spool root before invoking tools.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[GitHub_Mode_Failure_Modes]] · [[GitHub_Mode_Is_A_Different_Execution_Contract]]

16.4 GitHub-mode — Failure Modes

Breaking the GitHub-mode contract reintroduces agent risk. Direct vault writes bypass human mediation, making PKM corruption and unreviewable side effects likely. Ignoring vault-relative paths or deterministic spooling causes divergent outputs between local runs and CI, eroding trust in automation.

These failures collapse the safety boundary that GitHub-mode is meant to enforce.

Silent Partial Success

Mixed write semantics are more dangerous than hard failure. If a tool writes some outputs to the vault and others to the spool, the system enters an inconsistent state:

- **Vault changes**: Applied immediately, bypassing promotion review.
- **Spooled changes**: Staged for review, awaiting promotion.
- **Result**: The agent believes both changes succeeded. The human reviewer sees only the spooled outputs. The vault changes are invisible until discovered later—potentially after compounding errors.

This failure mode undermines the core GitHub-mode guarantee: all vault changes are human-mediated. Partial writes violate this by hiding some changes from the promotion workflow.

Hard failure is correct behavior: If the agent cannot write to the spool, the entire operation should fail visibly. This preserves system integrity and forces explicit handling of the error condition.

The no-vault-writes invariant ([[Book Zettel/MCP with DSPy Theory and Application/Execution Contexts/GitHub Mode/GitHub_Mode_No_Vault_Writes]]) exists to prevent silent partial success. Enforcement must be absolute—no exceptions for "safe" subfolders or "temporary" writes.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[GitHub_Mode_Why_This_Is_Not_Op · [[GitHub_Mode_Is_A_Different_Execution_Contract]]

16.5 GitHub-mode — Why This Is Not Optional

GitHub-mode removes direct write authority and routes changes through spooling so humans can review, regenerate, and validate before promotion. The guardrails keep long-running or autonomous tasks aligned, and they keep CI and GitHub Actions deterministic with the same path rules and spool root.

Without this contract, safety, reproducibility, and multi-agent correctness degrade immediately.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[Book Zettel/MCP with DSPy Theory and Application/Execution Contexts/GitHub Mode/GitHub_Mode_No_Vault_Writes]]

16.6 GitHub-mode Is a Different Execution Contract

GitHub-mode is NOT a restricted version of local mode. It is a distinct execution contract with a fundamentally different trust model.

Local Mode vs. GitHub-mode

In local mode, the agent writes directly to the vault. Human review is implicit—the user watches changes appear and manually rejects mistakes. This assumes synchronous human attention and full-time monitoring.

In GitHub-mode, the agent spools outputs to staging without touching the vault. Human review is explicit—changes are promoted only after approval. This enables unattended execution in CI, Actions, and scheduled workflows.

Vault Immutability Is a Correctness Guarantee

The vault write fence (D011) is not a convenience feature or an optimization. It is a correctness boundary. Without it:

- Agents can corrupt the PKM undetectably during batch or CI runs.
- Promotion workflows become meaningless—nothing left to review.
- The system violates its own contract: "All vault changes are human-mediated."

Trust Model Difference

Local mode trusts the user to monitor the agent continuously. GitHub-mode trusts the promotion workflow to mediate all changes.

These are incompatible assumptions. Conflating them erodes the guarantees that make GitHub-mode safe for automation.

No Backward Compatibility

GitHub-mode is not "local mode minus vault writes." Treating it as such invites unsafe optimizations. For example:

- "Just disable vault writes temporarily" → breaks determinism when absolute paths leak into outputs.
- "Make vault reads optional" → prevents agents from accessing necessary context.
- "Allow writes to specific subfolders" → introduces silent partial success (see [[GitHub_Mode_Failure_Modes]]).

GitHub-mode must be implemented as a distinct execution path, not a flag or runtime toggle.

Enforcement

All GitHub-mode guarantees must be enforced in code, not convention or policy. The execution context layer resolves which contract applies before any tool is invoked.

Links: [[MoC_Execution_Context_GitHub_Mode]] · [[Book Zettel/MCP with DSPy Theory and Application/Execution Contexts/GitHub Mode/GitHub_Mode_No_Vault_Writes]] · [[GitHub_Mode_Why_This_Is_Not_Optional]]

17

Chapter 15 Promotion to Skills and CI

17.1 When a Cluster Becomes a Skill

A cluster graduates once its core claims are runnable, verified, and reused by other workflows. Promotion is justified when the cluster is the smallest unit that reliably delivers value without manual curation.

Signals of graduation: - A stable interface exists (inputs, outputs, side effects). - The cluster can be tested end-to-end in CI. - Consumers depend on it as a contract, not as a narrative.

17.2 Skills as Executable Contracts

A skill is a contract that can run. The contract is defined by the behavior surface (inputs/outputs), and the execution proves the promise. The D009–D012 stance shifts from "document intent" to "execute intent".

Executable contracts: - Encode invariants in tests and fixtures. - Treat prompts and tool chains as versioned interfaces. - Make deviations visible before they reach production.

17.3 Skill Acceptance Criteria

Acceptance criteria define what it means for a skill to be "true". They are minimal, testable, and tied to observable outcomes. A promoted skill ships only when criteria are precise enough to automate.

Criteria checklist: - Defined happy path and explicit edge cases. - Deterministic inputs and measurable outputs. - Failure modes documented with guardrails.

17.4 CI as Enforcement, Not Validation

CI is the policy engine for promoted skills. It enforces contracts continuously rather than validating them once. The goal is to prevent drift, not to approve changes after the fact.

Enforcement stance: - Fail fast on contract violations. - Surface regressions in skill behavior, not just code coverage. - Treat flaky tests as skill instability.

17.5 Failure Modes When Skills Drift

Skill drift shows up as silent regressions between intent and execution. It happens when a skill's contract is stable but its behavior mutates.

Common failure modes: - Tests cover the old contract, not the new behavior. - Upstream data changes invalidate assumptions. - Tooling updates shift the execution surface without re-certification.

17.6 Promotion Gates and Graduation Checklist

Promotion gates define when a cluster can graduate to a skill. They pair D009– D012 intent with verifiable evidence.

Graduation checklist: - Contract and acceptance criteria committed. - CI suite enforces behavior across fixtures. - Owners and consumers agree on stability horizon. - Monitoring or retraining plan in place.

17.7 From Zettels to Skills: Operationalizing Architectural Intent

Zettels capture intent; skills operationalize it. Promotion turns architectural principles into runnable workflows with measurable outcomes. D009–D012 provide the rationale, while the skill package makes it enforceable.

Operationalization steps: - Distill the cluster into a single behavior contract. - Encode the contract in tests and CI gates. - Promote once the behavior is stable across scenarios.

18

Chapter 16: Agents as Junior Engineers

18.1 Agents Are Not Tools

Tools execute; agents reason

A tool performs a bounded action with predictable output. An agent interprets intent, makes choices, and can drift without guardrails, so it needs supervision beyond a command line.

Why the distinction matters

When you treat an agent like a deterministic tool, you skip review and context. The result is brittle automation that fails quietly and erodes trust.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_Are_Not_Peers]]
- [[CI_as_Supervision_Not_Validation]]

18.2 Agents Are Not Peers

The accountability gap

Agents can propose and execute steps, but they do not own outcomes in the way a human teammate does. Treating them as peers blurs responsibility and invites silent failure.

What peer-level work requires

Peer work depends on shared context, long-term judgment, and the ability to negotiate tradeoffs. Agents need explicit constraints and review to approximate that level of alignment.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_Are_Not_Tools]]
- [[Agents_as_Apprentices_Not_Autonomous_Actors]]

18.3 Agents as Apprentices, Not Autonomous Actors

The apprentice framing

Apprentices work with clear tasks, feedback loops, and incremental responsibility. Agents benefit from the same structure: explicit goals, bounded scope, and frequent check-ins.

Autonomy is earned, not assumed

Unsupervised autonomy amplifies misunderstandings. Treat autonomy as a graduation outcome after repeated, verified success.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_Are_Not_Peers]]
- [[Promotion_as_Trust_Graduation]]

18.4 Execution Contexts as Permission Levels

Contexts set the safety envelope

Execution contexts define what an agent can touch, from read-only analysis to write access. The context should match the risk profile of the task.

Aligning permissions with supervision

High-risk contexts require tighter review and smaller steps. Low-risk contexts can tolerate more autonomy, but still require visibility.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_as_Apprentices_Not_Autonomous_Actors]]
- [[CI_as_Supervision_Not_Validation]]

18.5 Skills as Onboarding Packets

Skills encode process

A skill should teach how work is done, not just what to run. It packages conventions, guardrails, and examples so agents learn the local workflow.

Onboarding over optimization

When skills are framed as onboarding packets, the agent becomes easier to supervise and more predictable across tasks.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Execution_Contexts_as_Permission_Levels]]
- [[Promotion_as_Trust_Graduation]]

18.6 CI as Supervision, Not Validation

CI is a feedback loop

Continuous integration should surface misalignment early, not serve as a last-minute approval gate. The goal is to catch drift while it is still small.

Human review stays in the loop

CI can verify invariants, but it cannot judge intent. Treat CI as a supervisor that flags issues for human confirmation.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Execution_Contexts_as_Permission_Levels]]
- [[Promotion_as_Trust_Graduation]]

18.7 Promotion as Trust Graduation

Trust is a sequence

Promotion means expanding scope only after repeated, verified success. Each step increases responsibility and the cost of mistakes.

Graduation criteria

Use clear signals: consistent task completion, reduced review overhead, and correct use of constraints. Without these, promotion is premature.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_as_Apprentices_Not_Autonomous_Actors]]
- [[CI_as_Supervision_Not_Validation]]

18.8 Failure Modes When Agents Are Treated Wrongly

Tool-style misuse

Treating agents like tools skips review and context checks, which leads to subtle errors that look like automation success until they compound.

Peer-style misuse

Treating agents like peers assumes judgment they do not have. This produces over-scoped tasks, vague requirements, and avoidable rework.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_Are_Not_Tools]]
- [[Agents_Are_Not_Peers]]

18.9 Why This Model Scales

Repeatable supervision

The junior-engineer model is scalable because it standardizes supervision: clear scopes, shared skills, and consistent review loops.

Trust grows without chaos

Graduated permissions and visible checkpoints allow more work to run in parallel without losing control or accountability.

Links

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Promotion_as_Trust_Graduation]]
- [[CI_as_Supervision_Not_Validation]]

19

Chapter 17: Supervision & Review Patterns

19.1 Chapter 17: Why Supervision Is Not Optional

Supervision is required because agent output cannot self-certify correctness or safety. The operator must validate results against explicit criteria and treat uncertainty as expected, not exceptional. This keeps accountability with humans and prevents silent drift.

Links

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]

19.2 Chapter 17: Shadow Execution and Dual-Run Patterns

Shadow execution keeps the agent in advisory mode while a human performs the actions. Dual-run compares the agent's output against a trusted baseline and focuses on deltas, not intent. Use these patterns when actions are irreversible or when new behaviors require validation.

Links

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]

19.3 Chapter 17: Review Gates and Human Signoff

Review gates define explicit checkpoints with evidence requirements before work is accepted. Human signoff confirms the criteria were met and assigns accountability for release. Gates prevent bypass and ensure acceptance is a deliberate decision.

Links

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]

19.4 Chapter 17: When Not to Automate

Do not automate when intent is ambiguous, effects are irreversible, or the system state cannot be observed. Refusal is the correct outcome when a human must clarify goals or accept responsibility for risk.

Links

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]

19.5 Chapter 17: Feedback Loops and Correction

Feedback loops capture failures as concrete artifacts and turn them into updated notes, checklists, or tests. Corrections must be evidence-driven and linked to the gate that now enforces them, closing the loop without relying on ad hoc prompts.

Links

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]

20

Chapter 18 — Prose Refinement Pipeline

20.1 Prose Refinement Objectives

- Improve clarity and precision without diluting technical accuracy.
- Align tone with APRESS/WROX-style technical prose.
- Preserve traceability: Zettels remain the source of truth.
- Keep edits atomic so changes are easy to review.

20.2 Prose Refinement Workflow

1. Identify candidate Zettels in the target chapter.
2. Edit in the vault (small, reviewable changes).
3. Record a short before/after summary.
4. Regenerate derived docs from the vault.
5. Rebuild book outputs (md/pdf/html/epub).
6. Review the chapter output and iterate.

20.3 Prose Refinement Checklist

☐ Single idea per paragraph.
☐ Consistent terminology (glossary alignment).
☐ Remove redundant phrasing and filler.
☐ Examples and code blocks match repo behavior.
☐ Regenerate derived docs and rebuild outputs.
☐ Log changes and rationale.

21

Chapter 99 — Agent Demo

21.1 Agent Demo Thesis

A minimal, safe MCP agent pipeline can create book-ready Zettels by enforcing contracts at tool boundaries and requiring explicit write intent.

21.2 Agent Demo Pipeline

1. Draft notes in `_agent_inbox` with `write_note`.
2. Stage cluster in repo `staging/`.
3. Promote to canonical book subtree.
4. Regenerate `docs/book` from vault MoCs.

21.3 Agent Demo Checklist

☐ MoC includes Acceptance Criteria
☐ Tags include `pkm` + `source`
☐ Source field present
☐ All links resolve
☐ Cluster ready for promotion

Appendix: Sources

Chapter 00 — Book Contract

- [[MoC_Chapter_00_Book_Contract]]
- [[Book Contract/MoC_Book_Contract]]
- [[Book Contract/What_Constitutes_a_Chapter]]
- [[Book Contract/Required_Chapter_Invariants]]
- [[Book Contract/Promotion_Lifecycle_of_Knowledge]]
- [[Book Contract/Failure_Modes_of_Large_Knowledge_Bases]]

Chapter 01: Foundations

- [[MoC_Chapter_01_Foundations]]
- [[Chapter_01_The_Problem_Space]]
- [[Chapter_01_What_is_MCP]]
- [[Chapter_01_Spec_Driven_Development]]
- [[Chapter_01_Why_DSPy_Matters]]
- [[Chapter_01_PKM_and_Agents]]
- [[Chapter_01_Foundations_Summary]]

Chapter 02: Constitution Layer

- [[MoC_Chapter_02_Constitution_Layer]]

- [[Chapter_02_Purpose_of_Agent_Constitution]]
- [[Chapter_02_Global_Principles_and_Guarantees]]
- [[Chapter_02_Safety_and_Compliance]]
- [[Chapter_02_Naming_and_Versioning_Conventions]]
- [[Chapter_02_Constitution_and_Spec_Interactions]]

Chapter 03: Spec-Driven Development Workflow

- [[MoC_Chapter_03_Spec_Driven_Development_Workflow]]
- [[Chapter_03_What_is_a_Spec]]
- [[Chapter_03_Anatomy_of_a_Spec]]
- [[Chapter_03_Spec_First_Workflow]]
- [[Chapter_03_Spec_to_MCP]]
- [[Chapter_03_Spec_to_DSPy]]
- [[Chapter_03_SDD_Feedback_Loop]]

Chapter 04: MCP Server Fundamentals

- [[MoC_Chapter_04_MCP_Server_Fundamentals]]
- [[Chapter_04_What_is_an_MCP_Server]]
- [[Chapter_04_MCP_Protocol_and_Sessions]]
- [[Chapter_04_Tool_Definitions_and_Schemas]]
- [[Chapter_04_MCP_Error_Models]]
- [[Chapter_04_Server_Capabilities_and_Versioning]]
- [[Chapter_04_State_and_Statelessness_in_MCP]]
- [[Chapter_04_MCP_with_DSPy_and_PKM]]

Chapter 05: DSPy Framework Deep Dive

- [[MoC_Chapter_05_DSPy_Framework_Deep_Dive]]
- [[Chapter_05_What_is_DSPy]]
- [[Chapter_05_DSPy_Signatures]]
- [[Chapter_05_DSPy_Modules]]
- [[Chapter_05_DSPy_Optimization]]
- [[Chapter_05_DSPy_orchestrating_MCP]]
- [[Chapter_05_DSPy_Memory_and_PKM]]
- [[Chapter_05_DSPy_Reasoning_Graphs]]
- [[Chapter_05_DSPy_as_PKM_Agent]]

Chapter 06: PKM Integration with Obsidian

- [[MoC_Chapter_06_PKM_Integration_with_Obsidian]]
- [[Chapter_06_PKM_in_Agent_Systems]]
- [[Chapter_06_Zettelkasten_as_Knowledge_Graph]]
- [[Chapter_06_Obsidian_as_PKM_OS]]
- [[Chapter_06_Designing_a_PKM_Vault]]
- [[Chapter_06_Zettel_Metadata_and_Schema]]
- [[Chapter_06_MCP_Tools_for_PKM]]
- [[Chapter_06_DSPy_and_PKM_Retrieval]]
- [[Chapter_06_PKM_as_Agent_Identity]]
- [[Chapter_06_PKM_MCP_DSPy_Loop]]

Chapter 07: MCP + DSPy + PKM Agent Project

- [[MoC_Chapter_07_MCP_DSPy_PKM_Agent_Project]]

- [[Chapter_07_What_is_an_Agent]]
- [[Chapter_07_Four_Layer_Agent_Architecture]]
- [[Chapter_07_PKM_Layer]]
- [[Chapter_07_DSPy_Layer]]
- [[Chapter_07_MCP_Layer]]
- [[Chapter_07_Constitution_Layer]]
- [[Chapter_07_Agent_Lifecycle]]
- [[Chapter_07_PKM_Agent_Reasoning_Graph]]
- [[Chapter_07_Agent_Failure_Modes_and_Recovery]]
- [[Chapter_07_Example_PKM_Query_Cycle]]

Full-Stack Example — `write_note`

- [[MoC_Full_Stack_Example_write_note]]
- [[Full_Stack_write_note_Why_First]]
- [[Full_Stack_write_note_Overwrite_Semantics]]
- [[Full_Stack_write_note_Specs_Tests_Examples_Form_a_Contract]]
- [[Full_Stack_write_note_From_Zettels_to_Skills_to_Code]]
- [[Full_Stack_write_note_What_To_Copy_For_Next_Tool]]

Full-Stack Example: search_notes (MoC)

- [[MoC_Full_Stack_Example_search_notes]]
- [[Full_Stack_search_notes_Why_Canonical_Read_Only]]
- [[Full_Stack_search_notes_Contract]]
- [[Full_Stack_search_notes_Tests_and_Examples]]
- [[Full_Stack_search_notes_Safety_Read_Only]]

- [[Full_Stack_search_notes_Skills_and_Constitution]]

Full-Stack Example — `update_note`

- [[MoC_Full_Stack_Example_update_note]]
- [[Full_Stack_update_note_Path_First_Safety_Model]]
- [[Full_Stack_update_note_Consent_and_Allowed_Roots]]

Full-Stack Example — `id_index_stats`

- [[MoC_Full_Stack_Example_id_index_stats]]
- [[Full_Stack_id_index_stats_Observability_and_Determinism]]
- [[Full_Stack_id_index_stats_D012_Vault_Relative_Paths]]
- [[Full_Stack_id_index_stats_Tool_Contract_and_Semantics]]
- [[Full_Stack_id_index_stats_Tests_and_Examples_Enforce_Contract]]
- [[Full_Stack_id_index_stats_Future_Id_Based_Mutation_Safety]]

Full-Stack Example — `append_to_note`

- [[MoC_Full_Stack_Example_append_to_note]]
- [[Full_Stack_append_to_note_Why_Append_Safer_Than_Overwrite]]
- [[Full_Stack_append_to_note_Consent_and_D011_Write_Fence]]
- [[Full_Stack_append_to_note_Relative_Paths_and_D012]]
- [[Full_Stack_append_to_note_Contract_and_Edge_Cases]]
- [[Full_Stack_append_to_note_Tests_and_Examples_Enforce_Contract]]
- [[Full_Stack_append_to_note_Journaling_Safety]]

Full-Stack Example: list_notes (MoC)

- [[MoC_Full_Stack_Example_list_notes]]

- [[Full_Stack_list_notes_Discovery_Companion]]
- [[Full_Stack_list_notes_Contract]]
- [[Full_Stack_list_notes_Safety_and_Scope]]
- [[Full_Stack_list_notes_Tests_and_Examples]]
- [[Full_Stack_list_notes_Hallucination_Mitigation]]
- [[Full_Stack_list_notes_Future_Navigation]]

Pattern: Journaling & Agent Memory (No New Tool)

- [[MoC_Journaling_Pattern]]
- [[Journaling_Pattern_Append_Only_Memory_Primitive]]
- [[Journaling_Pattern_Daily_Note_Naming_and_Location]]
- [[Journaling_Pattern_Create_If_Missing_and_Append_Safely]]
- [[Journaling_Pattern_Retrieval_Patterns]]
- [[Journaling_Pattern_Anti_Patterns]]
- [[Journaling_Pattern_Prepping_for_Append_Journal_Entry]]
- [[Journaling_Pattern_Temp_Vault_Walkthrough]]

Chapter 13 — Promotion & Compilation Pipeline

- [[MoC_Chapter_13_Promotion_Pipeline]]
- [[MoC_From_Zettels_to_Skills]]
- [[Zettels_Capture_Intent_Skills_Encode_Action]]
- [[Skills_as_Procedural_Zettels]]
- [[Why_Skills_Follow_Zettels_Not_Replace_Them]]
- [[Repository_as_Execution_Context_for_Skills]]

Execution Context: GitHub-mode

- [[MoC_Execution_Context_GitHub_Mode]]
- [[Book Zettel/MCP with DSPy Theory and Application/Execution Contexts/GitHub Mode/GitHub_Mode_No_Vault_Writes]]
- [[GitHub_Mode_Spool_Root_Semantics]]
- [[GitHub_Mode_Tool_Output_Invariants]]
- [[GitHub_Mode_Failure_Modes]]
- [[GitHub_Mode_Why_This_Is_Not_Optional]]
- [[GitHub_Mode_Is_A_Different_Execution_Contract]]

Chapter 15 Promotion to Skills and CI

- [[MoC_Chapter_15_Promotion_to_Skills_and_CI]]
- [[When_a_Cluster_Becomes_a_Skill]]
- [[Skills_as_Executable_Contracts]]
- [[Skill_Acceptance_Criteria]]
- [[CI_as_Enforcement_Not_Validation]]
- [[Failure_Modes_When_Skills_Drift]]
- [[Promotion_Gates_and_Graduation_Checklist]]
- [[From_Zettels_to_Skills_Operationalizing_Architectural_Intent]]

Chapter 16: Agents as Junior Engineers

- [[MoC_Chapter_16_Agents_as_Junior_Engineers]]
- [[Agents_Are_Not_Tools]]
- [[Agents_Are_Not_Peers]]
- [[Agents_as_Apprentices_Not_Autonomous_Actors]]
- [[Execution_Contexts_as_Permission_Levels]]

- [[Skills_as_Onboarding_Packets]]
- [[CI_as_Supervision_Not_Validation]]
- [[Promotion_as_Trust_Graduation]]
- [[Failure_Modes_When_Agents_Are_Treated_Wrongly]]
- [[Why_This_Model_Scales]]

Chapter 17: Supervision & Review Patterns

- [[MoC_Chapter_17_Supervision_and_Review_Patterns]]
- [[Chapter_17_Why_Supervision_Is_Not_Optional]]
- [[Chapter_17_Shadow_Execution_and_Dual_Run_Patterns]]
- [[Chapter_17_Review_Gates_and_Human_Signoff]]
- [[Chapter_17_When_Not_to_Automate]]
- [[Chapter_17_Feedback_Loops_and_Correction]]

Chapter 18 — Prose Refinement Pipeline

- [[MoC_Chapter_18_Prose_Refinement_Pipeline]]
- [[Prose_Refinement_Objectives]]
- [[Prose_Refinement_Workflow]]
- [[Prose_Refinement_Checklist]]

Chapter 99 — Agent Demo

- [[MoC_Chapter_99_Agent_Demo]]
- [[Agent_Demo_Thesis]]
- [[Agent_Demo_Pipeline]]
- [[Agent_Demo_Checklist]]