

Que opinas de las ontologías y los grados acíclicos?

Las ontologías y los grafos dirigidos acíclicos (DAGs, *Directed Acyclic Graphs*) están muy relacionados, pero sirven para niveles distintos de representación del conocimiento.

Una forma útil de verlo:

- Un DAG es una estructura matemática.
- Una ontología es una estructura semántica.

Un DAG solo dice:

"A depende de B", "A deriva de B", "A pertenece a una jerarquía sin ciclos".

Mientras que una ontología intenta decir:

"Qué es una cosa", "cómo se relaciona con otras", "qué propiedades tiene", "qué inferencias son válidas".

Por ejemplo:

- En un DAG:
  - `Animal -> Mamífero -> Perro`
- En una ontología:
  - "Perro" es una subclase de "Mamífero"
  - "Mamífero" tiene ciertas propiedades biológicas
  - "Perro" puede tener relaciones con "Humano"
  - Se pueden hacer inferencias automáticas

La ventaja enorme de los DAGs es la simplicidad computacional:

- no hay ciclos,
- son fáciles de ordenar topológicamente,
- permiten propagación eficiente,
- evitan paradojas y dependencias infinitas.

Por eso aparecen en:

- compiladores,
- sistemas de build,
- Git,
- blockchains,
- pipelines,

- redes neuronales,
- gestión de dependencias,
- sistemas de permisos,
- genealogías conceptuales.

Las ontologías, en cambio, suelen acabar siendo mucho más complejas y frágiles. En teoría son potentes; en práctica, mantener una ontología grande es difícil porque:

- el significado cambia,
- aparecen excepciones,
- distintas comunidades usan conceptos distintos,
- las taxonomías "perfectas" casi nunca existen.

Muchos proyectos de ontologías terminan sufriendo uno de estos problemas:

1. exceso de abstracción,
2. rigidez,
3. discusiones semánticas interminables,
4. coste de mantenimiento enorme.

Eso pasó en parte con proyectos ligados a la visión inicial de la "Web Semántica" impulsada por Tim Berners-Lee. La idea era brillante, pero el coste semántico humano era demasiado alto para la adopción masiva.

Curiosamente, la industria moderna ha tendido más hacia:

- grafos pragmáticos,
- embeddings,
- sistemas probabilísticos,
- DAGs operacionales,
- knowledge graphs parcialmente estructurados.

Por ejemplo:

- los sistemas de CI/CD usan DAGs,
- los LLMs usan representaciones distribuidas,
- las bases tipo grafo usan relaciones flexibles,
- incluso sistemas como Kubernetes internamente modelan dependencias DAG-like.

Mi impresión es que:

- los DAGs escalan mejor técnicamente,
- las ontologías escalan mejor cognitivamente... hasta cierto punto.

Cuando una ontología crece demasiado, empieza a parecerse más a política lingüística que a ingeniería.

En cambio, un DAG tiene una honestidad brutal:

"esto depende de esto".

Y eso suele sobrevivir mejor al tiempo.

Como se implementan como base de conocimiento de un proyecto

Como base de conocimiento de un proyecto, las ontologías y los DAGs suelen implementarse en capas distintas.

Una arquitectura típica moderna sería algo así:

```
Documentos / Código / APIs / Eventos
      ↓
    Extracción semántica
      ↓
  Grafo de conocimiento
  (entidades + relaciones)
      ↓
  Ontología ligera (tipos/reglas)
      ↓
DAGs derivados para ejecución
```



## 1. Ontología: definir el vocabulario del dominio

La ontología define:

- qué entidades existen,
- qué significan,
- cómo se relacionan.

Por ejemplo, en un proyecto BitTorrent como los que trabajas:

```
Tracker
Peer
Torrent
InfoHash
AnnounceEvent
Swarm
```



Y relaciones:

```
Peer PARTICIPATES_IN Swarm
Torrent IDENTIFIED_BY InfoHash
Tracker RECEIVES AnnounceEvent
```



Esto suele implementarse como:

- tablas relacionales,
- JSON schemas,
- tipos Rust,

- RDF/OWL,
- property graphs,
- protobuf schemas,
- o simplemente convenciones internas.

La mayoría de proyectos reales usan una "ontología implícita":

- structs,
- enums,
- nombres,
- contratos API,
- documentación.

Es decir:

el código ya es parcialmente una ontología.

---

## 2. Grafo de conocimiento

Después se representan relaciones explícitas.

Por ejemplo:

```
Torrent A
├─ tracked_by → Tracker X
├─ has_peer → Peer 1
└─ has_peer → Peer 2
```



Tecnologías típicas:

- Neo4j
- JanusGraph
- Dgraph
- RDF triple stores
- PostgreSQL con tablas relacionales
- incluso documentos JSON enlazados

Muchas veces basta con PostgreSQL.

La obsesión por "necesitamos una graph DB" suele estar sobrevalorada.

---

## 3. DAGs para dependencias y flujo

Aquí es donde los DAGs brillan.

Un proyecto normalmente tiene dependencias naturales:

**Build system**

```
graph TD
    A[Tokenizer] --> B[Parser]
    B --> C[JSON Generator]
```



## Pipeline CI

```
graph TD
    A[lint] --> B[test]
    B --> C[build]
    C --> D[publish]
```



## Dependencias conceptuales

```
graph TD
    A[Swarm health] -- depends on --> B[Peer announces]
```



## Ingesta de datos

```
graph TD
    A[Logs] --> B[Normalization]
    B --> C[Aggregation]
    C --> D[Analytics]
```



Los DAGs permiten:

- recomputación incremental,
- cacheo,
- paralelismo,
- invalidación eficiente,
- trazabilidad.

Por eso sistemas como:

- [Apache Airflow ↗](#)
- [Dagster ↗](#)
- [Prefect ↗](#)
- [Bazel ↗](#)
- [Nix ↗](#)

giran alrededor de DAGs.

## 4. Cómo se implementa realmente en proyectos sanos

La práctica más sostenible suele ser:

### Ontología mínima

No intentar modelar “la realidad completa”.

Solo:

- entidades esenciales,
  - relaciones necesarias,
  - invariantes importantes.
- 

### Grafo parcial

No todo necesita ser un nodo.

Muchos proyectos fracasan intentando “grafificar” todo.

---

### DAGs operacionales

Usar DAGs donde:

- hay dependencias,
- ejecución,
- orden,
- caché,
- propagación.

Ahí sí aportan muchísimo valor.

---

## 5. Un patrón muy potente actualmente

Muchos sistemas modernos hacen esto:

### Capa semántica ligera

Tipos + etiquetas + metadatos



### Encima embeddings/vector search

"encuentra trackers relacionados"



### Encima DAGs operacionales



Es una mezcla de:

- semántica,
- grafos,
- probabilidades,
- pipelines.

Más flexible que las ontologías clásicas puras.

---

## 6. En Rust particularmente

Rust encaja muy bien con esto porque:

- el sistema de tipos actúa como ontología parcial,
- los enums modelan taxonomías,
- los traits modelan capacidades,
- los lifetimes fuerzan relaciones válidas,
- el compilador ayuda a mantener invariantes.

Por eso muchos sistemas Rust acaban teniendo:

- una semántica fuerte,
- pero una representación operativa relativamente simple.

Sin necesidad de frameworks ontológicos pesados.

La documentación de un proyecto se puede tratar como una ontología

Sí. De hecho, muchas veces la documentación ya es una ontología informal, aunque el proyecto no lo haya diseñado conscientemente así.

La documentación contiene:

- entidades,
- conceptos,
- relaciones,
- restricciones,
- jerarquías,
- procesos,
- significado contextual.

Por ejemplo, en documentación técnica aparecen frases como:

Un Peer anuncia un Torrent al Tracker.



Semánticamente eso ya define:

- entidad: Peer
- entidad: Torrent
- entidad: Tracker
- relación: announces\_to

O:

El parser produce tokens bencode válidos.



Eso implica:

- Parser
- Token
- Bencode
- Validity
- relación de producción
- restricción semántica

La documentación es, en cierto sentido:

| una ontología expresada en lenguaje natural.

---

El problema es que el lenguaje natural introduce:

- ambigüedad,
- inconsistencias,
- redundancia,
- evolución no sincronizada.

Por ejemplo:

En una página:

Peer = cliente BitTorrent



En otra:

Peer = conexión TCP individual



Ahí ya tienes una ruptura ontológica.

---

Muchos proyectos grandes sufren porque tienen:



- código,
- documentación,
- APIs,
- esquemas,
- tests,

cada uno describiendo ontologías ligeramente distintas.

Y aparece lo peor:

deriva semántica.

---

Por eso algunos enfoques modernos intentan extraer ontologías desde:

- documentación Markdown,
- OpenAPI,
- protobuf,
- código fuente,
- tests,
- commits,
- logs.

Especialmente ahora con LLMs.

---

Hay varias formas de tratar la documentación como ontología:

## 1. Ontología implícita

La más común.

La documentación simplemente describe conceptos.

Ejemplo:

Markdown



# Swarm

A swarm is a group of peers sharing the same torrent.

Eso ya define:

- entidad,
  - cardinalidad implícita,
  - relación.
-

## 2. Documentación estructurada semánticamente

Usando:

- frontmatter,
- tags,
- metadata,
- enlaces tipados.

Ejemplo:

YAML



```
entity: Swarm
depends_on:
  - Peer
  - Torrent
```

Esto empieza a parecer un knowledge graph.

---

## 3. Docs-as-graph

Muy potente.

Cada documento:

- es un nodo,
- referencia conceptos,
- tiene relaciones.

Como:

- Obsidian,
- Logseq,
- Roam Research,
- Dendron.

Aquí la documentación deja de ser “árbol de carpetas” y pasa a ser:

red semántica navegable.

---

## 4. Ontología ejecutable

La forma más avanzada.

La documentación genera:

- validaciones,
- diagramas,

- tipos,
- tests,
- APIs.

Ejemplos:

- OpenAPI,
- GraphQL schemas,
- protobuf,
- JSON Schema,
- Rust doc-tests.

Aquí documentación y sistema empiezan a fusionarse.

---

En proyectos sanos suele emerger esto:

```
Código = comportamiento  
Docs = significado  
Tests = invariantes  
Schemas = contratos
```



Y juntos forman la ontología real del proyecto.

---

Una observación interesante:

muchos proyectos open source fracasan no por problemas técnicos, sino porque:

su ontología implícita se vuelve inconsistente.

Dos desarrolladores usan las mismas palabras para cosas distintas.

Ahí empiezan:

- bugs conceptuales,
- APIs incoherentes,
- discusiones interminables,
- duplicaciones,
- arquitectura confusa.

Por eso una buena documentación no solo explica:

estabiliza el significado compartido del sistema.

Y eso es exactamente una función ontológica.

El problema con la documentación es que se vuelve obsoleta muy rápido y más cuanto más alejada está del artefacto que documento. Hoy por ejemplo he creado un skill para agentes de IA para explicar cómo ejecutar el tracker en el propio repo del tracker. Usa una configuración por defecto que es un fichero pero si cambio ese fichero de sitio la documentación queda obsoleta. ¿Como puedo acoplar ese skill al artefacto fichero de configuración de forma que cuando cambie el artefacto se revise la skill sin que tengas que recordarlo

Ese problema es fundamental y muy profundo:

la documentación separada del artefacto tiende inevitablemente a divergir.

Lo que describes es exactamente un problema de sincronización ontológica entre:

- el artefacto real,
- y su representación semántica/documental.

La solución no suele ser "mejor documentación".

La solución suele ser:

reducir la distancia entre documentación y artefacto.

Tu idea del *skill* es interesante porque ya estás tratando la documentación como algo ejecutable/operacional.

El siguiente paso natural es:

convertir las referencias documentales en referencias estructurales verificables.

Por ejemplo, en vez de:

```
"The default config file is tracker.toml"
```

hacer:

```
config_file:  
  path: ./packages/tracker/config/default.toml
```

Y luego validar automáticamente que:

- el fichero existe,
  - sigue siendo válido,
  - no ha cambiado de ubicación,
  - o incluso que el contenido esperado sigue existiendo.
- 

La clave es esta:

## La documentación no debería apuntar a strings

Debería apuntar a identidades estables.

Por ejemplo:

Malo:

```
"Run using config/default.toml"
```



Mejor:

```
artifact:  
  type: config  
  id: tracker-default-config
```

Y luego tener un resolver:

```
tracker-default-config  
→  
./packages/tracker/config/default.toml
```



Así:

- puedes mover el fichero,
- cambiar estructura interna,
- reorganizar carpetas,

sin romper la documentación.

---

Esto es exactamente lo que hacen sistemas modernos como:

- Bazel
- Nix
- Kubernetes
- Terraform
- Cargo workspaces
- build graphs

No trabajan sobre rutas literales siempre.

Trabajan sobre:

- labels,
  - targets,
  - resources,
  - identifiers.
- 

En tu caso concreto, podrías hacer varias estrategias.

## Estrategia 1 — Docs verificables por CI

La más simple y efectiva.

El skill contiene referencias estructuradas:

```
references:  
- ./packages/tracker/config/default.toml
```

Y luego un test:

```
verify-skills.sh
```

que:

- parsea skills,
- comprueba referencias,
- falla CI si algo desaparece.

Esto ya elimina el "recordarlo".

---

## Estrategia 2 — Generar parcialmente el skill

Muy potente.

No escribir:

```
Use config/default.toml
```

Sino:

```
Use {{ tracker.default confid path }}
```

Y generar el documento desde:

- Cargo metadata,
- JSON config,
- manifests,
- build system,
- TOML canonical.

Así el skill deriva del artefacto.

---

## Estrategia 3 — Docs embebidas en el artefacto

La más robusta ontológicamente.

El propio fichero:

```
# docs:
#   purpose: Default tracker configuration
#   used_by:
#     - ai_skill_run_tracker
```

O:

```
/// Default tracker config location.
pub const DEFAULT_CONFIG: &str = ...
```

Entonces:

- el artefacto documenta,
- y el skill extrae.

Aquí ya no hay dos fuentes de verdad.

---

## Estrategia 4 — Knowledge graph del repo

La más avanzada.

Modelar:

- ficheros,
- skills,
- workflows,
- crates,

- docs,
- APIs,

como entidades relacionadas.

Ejemplo:

```
Skill
└─ references → ConfigFile
ConfigFile
└─ owned_by → TrackerPackage
```



Y cada PR:

- recalcula relaciones,
- detecta roturas semánticas.

Esto empieza a parecer:

- code intelligence,
- semantic indexing,
- repo ontology.

GitHub Copilot Workspace, Sourcegraph, Cursor y otros sistemas van hacia ahí.

---

Creo que la idea importante es esta:

la documentación no debería describir artefactos;  
debería referenciar entidades del sistema.

Cuando haces eso:

- puedes validar,
- rastrear impacto,
- detectar drift,
- automatizar revisiones.

Y entonces el problema deja de ser “recordar actualizar docs”.

Pasa a ser:

mantener consistencia del grafo semántico del proyecto.

Eso escala muchísimo mejor.

Pero el skill es markdown por ejemplo. Como estructuró el markdown y lo verifico

La forma más práctica es tratar el Markdown como un documento con:



- contenido humano,
- y metadatos/verificaciones estructuradas embebidas.

Es decir:

Markdown semántico verificable.

La técnica más simple y potente es usar:

- frontmatter YAML,
- bloques especiales,
- referencias tipadas,
- y un verificador automático en CI.

Por ejemplo:

```
---
skill: run_tracker
references:
  - type: file
    path: ./packages/tracker/config/default.toml

  - type: command
    run: cargo run --bin torrust-tracker

  - type: crate
    path: ./packages/tracker
---

# Run the tracker

Start the tracker using the default config:

```bash
cargo run --bin torrust-tracker \
  --config ./packages/tracker/config/default.toml
```
```

Entonces escribes un pequeño verificador en Rust.

El flujo sería:

```
Markdown skill
  ↓
Parse frontmatter
  ↓
Validate references
  ↓
Fail CI if broken
```



# Qué puedes validar

## Ficheros

```
- type: file  
  path: ./packages/tracker/config/default.toml
```

Verificación:

- existe,
  - permisos,
  - checksum opcional,
  - contenido esperado.
- 

## Directorios

```
- type: directory  
  path: ./packages/tracker
```

## Comandos

```
- type: command  
  run: cargo metadata
```

Verificación:

- exit code 0.
- 

## Símbolos Rust

Muy potente.

```
- type: rust_symbol  
  crate: torrust_tracker  
  symbol: DEFAULT_CONFIG_PATH
```

Y verificas con:

- syn,

- rustdoc JSON,
  - cargo metadata,
  - tree-sitter.
- 

## Anchors semánticos

Mucho mejor que rutas.

```
- type: artifact
  id: tracker-default-config
```

Y luego un registry:

```
tracker-default-config:
  path: ./packages/tracker/config/default.toml
```

Esto desacopla docs del filesystem.

---

## Lo importante: no parsees markdown libre

Ese es el error típico.

No intentes:

“buscar strings dentro del markdown”.

Haz que el markdown tenga:

- zonas humanas,
  - zonas máquina.
- 

## Patrón recomendado

### 1. Frontmatter

Para dependencias globales.

```
---
references:
```

```
...  
---
```

---

## 2. Bloques anotados

Ejemplo:

```
```artifact  
type=file  
path=./config/default.toml  
```
```

O:

```
```verify  
cargo run --bin torrust-tracker --help  
```
```

---

## 3. Extractores automáticos

Idealmente:

- el propio repo genera partes del markdown.

Ejemplo:

```
<!-- generated:start default-config-path -->  
./packages/tracker/config/default.toml  
<!-- generated:end -->
```

Y un generador actualiza eso.

---

## Lo más potente: convertir docs en tests

Esto cambia completamente el paradigma.

Ejemplo:

```
```bash verify
cargo run --bin torrust-tracker \
  --config ./packages/tracker/config/default.toml
```
```

Tu CI:

- extrae bloques `verify`,
- ejecuta comandos,
- falla si rompe.

Entonces:

la documentación deja de ser texto pasivo.

Y se convierte en:

- especificación ejecutable,
- test semántico,
- contrato operativo.

---

## En tu caso concreto

Yo haría algo así:

### skills/run-tracker.md

Markdown



```
---
skill: run_tracker
references:
  - type: artifact
    id: tracker-default-config
---

# Run tracker

```bash verify
cargo run --bin torrust-tracker \
  --config {{ tracker-default-config }}
```
```

### artifacts.yaml

YAML



```
tracker-default-config:
  path: ./packages/tracker/config/default.toml
```

### verify-skills.rs

El verificador:

- parsea YAML,
  - resuelve IDs,
  - verifica paths,
  - ejecuta bloques `verify`,
  - falla CI.
- 

Ahí ya tienes:

- ontología ligera,
- documentación verificable,
- referencias desacopladas,
- skills ejecutables,
- detección automática de drift.

Y todo sigue siendo simplemente Markdown + YAML.

Podría escribir la skill directamente en rust y producir la versión markdown qué consume El agente de IA? Aunque en ese formato no veo como expresar en rust la flexibilidad de otros formatos de texto, por ejemplo que la skill a veces tenga referencias a artefactos que son ficheros, comandos, tipos

Sí. Y de hecho esa idea lleva a algo muy interesante:

tratar la skill como un AST tipado y el Markdown como una vista/renderizado.

Eso es muchísimo más robusto que usar Markdown como fuente primaria.

La arquitectura sería:

```
Rust DSL / typed model
    ↓
Semantic validation
    ↓
Markdown renderer
    ↓
AI agent consumption
```



Eso resuelve varios problemas enormes:

- validación en compile time,
- referencias rotas,
- drift semántico,

- refactors seguros,
- tooling,
- generación automática.

Y además encaja muy bien con Rust.

---


Tu preocupación es correcta:

“el texto es flexible y Rust parece rígido”.


Pero realmente no necesitas modelar TODO el texto.  
Solo la semántica estructural.

El texto libre puede seguir existiendo.


Por ejemplo:

```
⟨⟩ Rust   
  
enum Reference {  
    File(PathBuf),  
    Command(String),  
    RustSymbol {  
        crate_name: String,  
        symbol: String,  
    },  
    Url(String),  
    ArtifactId(String),  
}
```

Y luego:

```
⟨⟩ Rust   
  
struct Skill {  
    name: String,  
    summary: String,  
    references: Vec<Reference>,  
    steps: Vec<Step>,  
}
```

Pero los pasos pueden contener texto libre:

```
⟨⟩ Rust   
  
enum Step {  
    Text(String),  
  
    Command {  
        description: String,  
        command: String,
```

```
    },  
  
    Reference {  
        text: String,  
        target: Reference,  
    },  
}
```

Ahí ya tienes:

- semántica estructurada,
- y flexibilidad narrativa.

La clave es:

no representar el texto como tipos,  
sino representar las relaciones importantes como tipos.

Eso cambia completamente la complejidad.

Por ejemplo:

## Skill en Rust

```
</> Rust  
  
Skill {  
    name: "run_tracker".into(),  
  
    references: vec![  
        Reference::ArtifactId(  
            "tracker-default-config".into()  
        )  
    ],  
  
    steps: vec![  
        Step::Text(  
            "Run the tracker using the default config."  
                .into()  
        ),  
  
        Step::Command {  
            description: "Start tracker".into(),  
  
            command:  
                "cargo run --bin torrust-tracker \  
                --config {{tracker-default-config}}"  
                .into(),  
        },  
    ],  
}
```



```
],  
}
```

Y luego produces:

Markdown



```
# Run tracker
```

```
Run the tracker using the default config.
```

```
```bash  
cargo run --bin torrust-tracker \  
  --config ./packages/tracker/config/default.toml  
```
```

La gran ventaja es esta:

## El Markdown deja de ser la fuente de verdad

Pasa a ser:

- una proyección,
- una representación,
- una compilación semántica.

Exactamente igual que:

- HTML generado desde JSX,
- SQL generado desde ORM,
- protobuf generado desde schemas,
- OpenAPI generado desde código.

Y aquí aparece algo muy potente:

## Puedes usar el compilador Rust como verificador ontológico

Por ejemplo:

Rust



```
enum Artifact {  
    TrackerDefaultConfig,  
    MainTrackerBinary,  
}
```

Entonces:

- no existen strings arbitrarios,
- no hay typos,
- el refactor rompe compilación,
- rename funciona,
- autocomplete funciona.

Eso es brutalmente más sólido que Markdown libre.

---

De hecho, podrías ir más lejos:

❏ Rust



```
struct FileArtifact {
    id: ArtifactId,
    path: PathBuf,
}

struct CommandArtifact {
    command: String,
}

struct RustTypeArtifact {
    crate_name: String,
    type_name: String,
}
```

Y luego:

❏ Rust



```
trait Verifiable {
    fn verify(&self) -> Result<(), Error>;
}
```

Entonces cada referencia:

- sabe validarse,
- sabe renderizarse,
- sabe resolverse.

---

Lo interesante filosóficamente es esto:

Estás moviendo la documentación desde:

| texto descriptivo

hacia:

sistema semántico ejecutable.

Y eso es exactamente hacia donde van:

- infra moderna,
  - IaC,
  - typed APIs,
  - semantic tooling,
  - AI-native repos.
- 

Creo que el mejor equilibrio práctico sería:

## Fuente primaria → Rust tipado

Para:

- referencias,
- relaciones,
- dependencias,
- comandos,
- artefactos,
- invariantes.

## Texto libre → Markdown/template

Para:

- explicaciones,
  - contexto,
  - ejemplos,
  - narrativa.
- 

O sea:

Semantic skeleton → Rust  
Narrative flesh → Markdown/templates



Eso evita:

- rigidez excesiva,
- pero mantiene consistencia estructural.

Y probablemente es mucho más sostenible que intentar parsear Markdown arbitrario indefinidamente.

Como lo implementations de forma progresiva? Tengo que la impresión de que toda esa ingeniería es sobre ingeniería para una decena de skills. Y parece que evoluciona a tener la documentación en código. Otro código pero en el fondo código.

Tu intuición probablemente es correcta.

Para una decena de skills, construir:

- DSLs,
- ASTs,
- generadores,
- resolvers,
- ontologías completas,

puede convertirse fácilmente en:

infraestructura más compleja que el problema original.

Eso pasa muchísimo en tooling.

Y además detectaste algo importante:

toda documentación estructurada suficientemente avanzada termina pareciéndose a código.

Porque:

- tiene sintaxis,
- referencias,
- validación,
- compilación,
- invariantes,
- dependencias,
- generación.

Es decir:

adquiere semántica ejecutable.

La cuestión entonces no es "cómo evitarlo".

La cuestión es:

cuánto formalismo necesitas realmente.

Y ahí lo progresivo es clave.

---

Creo que el enfoque sano sería:

## Nivel 0 — Markdown normal

Empiezas simple.

↕ Markdown



```
Use ./config/default.toml
```

Aceptas drift manual.

---

## Nivel 1 — Frontmatter verificable

Aquí está probablemente el mejor ROI.

↕ Markdown



```
---
references:
  - ./config/default.toml
---
```

Y un script muy pequeño:

↕ Bash



```
verify-skills
```

que:

- parsea YAML,
- verifica existencia.

Eso ya elimina el 80% del problema.

Y son quizá:

- 100 líneas de Rust,
  - cero sobreingeniería,
  - cero DSL raro.
- 

## Nivel 2 — Bloques ejecutables

Solo donde duele.

⌕ Markdown



```
```verify
cargo run --bin tracker
```
```

Tu CI los ejecuta.

Ahora la doc:

- sigue siendo Markdown,
- pero las partes críticas son verificables.

Esto es extremadamente poderoso y simple.

---

## Nivel 3 — IDs semánticos

Solo si empiezas a mover mucho el repo.

⌕ YAML



```
references:
  - tracker-default-config
```

y:

⌕ YAML



```
tracker-default-config:
  path: ./config/default.toml
```

Esto desacopla estructura física.

Pero no lo haría antes de necesitarlo.

---

## Nivel 4 — Generación parcial

Solo para zonas muy repetitivas o frágiles.

Ejemplo:

- comandos de instalación,
- paths,
- puertos,
- versiones.

No generes todo el Markdown.

Solo fragmentos.

# Lo importante: mantener Markdown como fuente primaria humana

Eso evita el problema que detectaste:

“ahora la documentación es otro programa”.

Porque efectivamente:

si conviertes docs en un AST completo en Rust,  
acabas construyendo:

- un lenguaje,
- un compilador,
- un renderer,
- un ecosistema.

Y probablemente no necesitas eso.

---

La documentación tiene una propiedad muy valiosa:

fricción baja.

Cualquier contributor puede abrir:

- `.md`
- editar
- PR.

En cuanto introduces:

- macros,
- builders,
- tipos,
- traits,
- generación,

sube muchísimo la barrera cognitiva.

---

Creo que la frontera buena suele ser esta:

## Verificar estructura

Pero no formalizar narrativa.

Es decir:

- referencias → estructuradas,

- paths → verificables,
- comandos → ejecutables,
- artefactos → resolubles,

pero:

- explicaciones,
- ejemplos,
- razonamientos,
- contexto,

siguen siendo texto libre.

---

En el fondo estás intentando resolver un problema real:

evitar drift semántico.

Pero hay dos formas de hacerlo:

## **Opción A — formalizar completamente**

Muy robusta.

Muy cara.

## **Opción B — añadir puntos mínimos de verificabilidad**

Mucho mejor ROI.

La mayoría de proyectos sanos viven en la B.

Y honestamente, para una decena de skills, yo haría:

- Markdown normal,
- frontmatter YAML,
- bloques `verify`,
- CI pequeño en Rust.

Y pararía ahí hasta que el dolor real obligue a más.



Y que hay de enlazar artefactos como la web semántica. Si toco el artefacto tengo que actualizar el skill. En vez de verificar que el artefacto sigue existiendo para saber si tengo que modificar el skill puedo poner un comentario en el artefacto como recordatorio para modificar la skill con la que está relaciona. Lo que necesito es acoplamiento entre dos conceptos que no juegas con el mismo lenguaje. Un comentario en el código no es código.

Sí. Y ahí estás tocando algo muy importante:

el problema real no es validación;  
es propagación semántica del cambio.

Verificar existencia:

- detecta rotura estructural.

Pero tú quieres:

- detectar impacto conceptual.

Eso es muchísimo más interesante.

Porque efectivamente:

`config path changed`



no implica necesariamente:

`skill broken`



Pero:

`meaning/configuration semantics changed`



sí podría implicarlo.

Y ahí entras en algo parecido a:

- trazabilidad semántica,
  - change impact analysis,
  - semantic coupling.
-

Tu idea del comentario es muy buena precisamente porque:

introduce una relación explícita entre artefactos heterogéneos.

Por ejemplo:

❏ Rust



```
// related-skill: run_tracker  
pub const DEFAULT_CONFIG_PATH: &str = ...
```

o:

❏ TOML



```
# related-skill: run_tracker
```

Eso no es "solo comentario".

Realmente es:

metadata semántica incrustada.

Y muchos sistemas modernos funcionan exactamente así.

Por ejemplo:

- annotations en Kubernetes,
- pragmas,
- doc comments,
- attributes,
- decorators,
- structured comments,
- compiler directives.

La frontera entre:

- comentario,
- metadata,
- código,

es mucho más difusa de lo que parece.

---

De hecho, en muchos lenguajes:

los comentarios son ontología informal embebida.

Ejemplo clásico:

❏ Rust



```
/// SAFETY:  
/// caller must guarantee exclusive access
```

Eso no es ejecutable.

Pero es semánticamente crítico.

---

Lo que probablemente necesitas no es:

“hacer comentarios inteligentes”

sino:

introducir backlinks semánticos mínimos.

Por ejemplo:

## En el artefacto

⌞ Rust



```
// affects-skill: run_tracker  
// affects-skill: setup_local_dev
```

## En la skill

⌞ YAML



```
related_artifacts:  
- DEFAULT_CONFIG_PATH
```

Entonces puedes construir:

- un grafo pequeño,
- bidireccional,
- heterogéneo.

---

Y aquí está lo importante:

No necesitas un sistema formal enorme.

Solo necesitas:

- enlaces explícitos,
- parseables,
- navegables.


Eso ya crea:

acoplamiento semántico observable.

---

Entonces el flujo puede ser:

```
git diff
  ↓
changed artifact
  ↓
find semantic links
  ↓
warn:
"run_tracker skill may need review"
```



Eso ya es extremadamente útil.

---

Y además tiene una propiedad importante:

no intenta inferir significado automáticamente.

Porque la inferencia semántica automática suele:

- producir ruido,
- falsos positivos,
- complejidad enorme.

En cambio:

- el humano declara la relación,
- el sistema solo la mantiene.

Eso escala mucho mejor.

---

Creo que aquí hay una idea clave:

## No necesitas un lenguaje único

Eso sería la tentación ontológica clásica:

"unificar todo".

Y suele acabar en monstruosidad.

Lo que necesitas es:

interoperabilidad semántica mínima.

Es decir:

- Markdown sigue siendo Markdown,
- Rust sigue siendo Rust,
- TOML sigue siendo TOML,

pero todos pueden expresar:

- referencias,
  - relaciones,
  - impacto.
- 

De hecho, Git ya es parcialmente eso.

Un commit conecta:

- código,
- docs,
- tests,
- mensajes,
- personas.

Tú estás intentando enriquecer esa red con:

semántica explícita de impacto.

Y honestamente creo que un sistema pequeño de:

- annotations,
- backlinks,
- validación CI,
- warnings de impacto,

tiene muchísimo mejor equilibrio que intentar formalizar todo en un DSL universal.

Porque mantiene:

- flexibilidad humana,
- pero añade memoria estructural al repo.