doc_pipeline

Text → Architecture → Prompt Engineering Engine

Ghost in the Machine Labs · March 2026 · Rev 1.0

A three-script pipeline that converts any specification document into a clean concept graph, then synthesizes targeted prompts for every subsystem. Feeds directly into RM's geometric substrate.

1. Overview

doc_pipeline converts unstructured text — architecture docs, feature specs, decision records, phase descriptions — into structured concept triples and from there into actionable prompts. It is the translation layer between human language and RM's geometric substrate.

Script	Purpose
doc_to_triples.py	`Parse .docx → concept graph (subject, relation, object)`
triple_polish.py	`Dialog loop → polish triples to 100% quality (unity)`
triples_to_prompt.py	`Concept graph → structured prompt in 3 modes`

The three scripts are a pipeline. The output of each feeds the next. Any specification document can enter at step one and exit as a precision prompt.

2. Pipeline Architecture

The full pipeline:

```
DOCUMENT (.docx)
     │
     ▼  doc_to_triples.py
CONCEPT GRAPH  ──────────────────────────────────────
{stem}_triples.json   — (subject, relation, object) set
{stem}_vocab.json     — Edinburgh association pairs for RM
{stem}_queries.json   — structural query battery
     │
     ▼  triple_polish.py
POLISHED GRAPH (unity = 100% quality)
{stem}_triples_polished.json
{stem}_polish_report.json
{stem}_unresolved.json  ← empty if unity achieved
     │
     ▼  triples_to_prompt.py
STRUCTURED PROMPT
Mode: task  — ordered steps, I/O contracts, constraints
```

```
  Mode: llm   — full system prompt with context + antipatterns
  Mode: rm    — terse geometric query for RM substrate
```

> **Design Principle**
> Each script is standalone and composable. You can enter the pipeline at any stage. If you already have a clean triple set, run triples_to_prompt.py directly. If you have a polished triple set from a previous session, skip extraction entirely.

3. Script Reference

3.1 doc_to_triples.py

Parses a .docx file and extracts a structured concept graph.

Usage

```
python3 doc_to_triples.py <document.docx> [--out-dir DIR] [--verbose]

# Examples
python3 doc_to_triples.py spec.docx
python3 doc_to_triples.py spec.docx --out-dir ./output --verbose
```

Outputs

File	Contents
{stem}_triples.json	(subject, relation, object) concept graph with source attribution
{stem}_vocab.json	Edinburgh-format association pairs for RM substrate load
{stem}_queries.json	Completeness, gap, and archetype query battery

Relation Types Extracted

```
is_type          has_component    depends_on       produces
consumes         transforms_to    implements       extends
executes         validates        stores           loads
calls            returns          has_property     has_constraint
precedes         follows          maps_to          defined_by
governed_by      owned_by
```

> **What it parses**
> Heading hierarchy, body paragraphs, bullet label:description pairs, tables (row-to-triple mapping), and function signatures. The more structured your document, the richer the graph.

3.2 triple_polish.py

Dialog improvement loop. Scores every triple, applies automatic repairs, then runs interactive dialog for items that require human input. Loops until 100% quality (unity) or you type 'done'.

Usage

```
python3 triple_polish.py <triples.json> [--auto] [--max-rounds N]

# Interactive (default) — loops until unity
python3 triple_polish.py output/spec_triples.json

# Auto-only — apply mechanical repairs, skip dialog
python3 triple_polish.py output/spec_triples.json --auto

# Cap at 3 rounds
python3 triple_polish.py output/spec_triples.json --max-rounds 3
```

Quality Scoring

Each triple receives a score from 0.0 (noise) to 1.0 (clean atomic concept). The loop targets 100%.

Failure Modes — Auto-Repaired

Mode	Description & Fix
HEADING_NUM	Section number prefix canonicalized as concept (e.g. "13_foo"). Strip prefix.
PROSE_SHARD	Full sentence absorbed as object token. Truncate to 2 content words.
ANTI_PATTERN	Negative instruction encoded as positive concept. Re-encode as violates triple.

Failure Modes — Dialog Required

Mode	Dialog Format
CLAUSE_FRAG	subject=<token> object=<token> | discard | skip
PRONOUN	subject=<token> | discard | skip
MISSING_IFACE	contract=<token> | skip

Outputs

File	Contents
{stem}_triples_polished.json	Clean triple set — ready for prompt synthesis or RM load
{stem}_polish_report.json	Full audit: repairs per round, quality %, failure summary
{stem}_unresolved.json	Remaining triples below unity (empty if 100% achieved)

> **Unity**
> Unity means every active triple scores 1.0. The unresolved.json file is empty at unity. The polished triple set is then suitable for both triples_to_prompt.py synthesis and direct RM substrate load.

3.3 triples_to_prompt.py

Synthesizes structured prompts from a polished concept graph. Walks the graph outward from a target token and assembles output in one of three modes.

Usage

```
python3 triples_to_prompt.py <triples.json> <target> [--mode MODE] [--depth N]

# List available tokens
python3 triples_to_prompt.py output/spec_polished.json --list

# Task spec (default)
python3 triples_to_prompt.py output/spec_polished.json pipeline_phase_1

# LLM system prompt
python3 triples_to_prompt.py output/spec_polished.json architectural_principles --mode llm

# RM substrate query
python3 triples_to_prompt.py output/spec_polished.json gpu_resource_management --mode rm

# Write to file
python3 triples_to_prompt.py output/spec_polished.json bootstrap_language --out prompt.txt

# Generate prompts for every top-level token (batch)
python3 triples_to_prompt.py output/spec_polished.json --all-tokens --mode task
```

Output Modes

Mode	Flag	Output
Task Spec	`--mode task`	Ordered steps, I/O contracts, constraints, failure modes, acceptance criteria
LLM Prompt	`--mode llm`	Full system prompt: role, context, input/output contracts, antipatterns, success criteria
RM Query	`--mode rm`	Terse geometric relation block for RM substrate. No prose. Hub-filtered.

Graph Traversal

--depth controls how many hops from the target token are included. Default is 2. Depth 1 = direct relations only. Depth 3+ = full neighborhood including transitive dependencies.

> **Token Resolution**
> Target tokens are matched by substring if an exact match is not found. "phase_1" will resolve to "pipeline_phase_1_domain_triangulation" if that is the only match. Use --list to see all available tokens.

4. Usage Scenarios

4.1 Text to Architecture

Convert a loose natural-language specification into a structured architecture.

1. Write your spec in plain English in a .docx file. Use headings for major components, bullets for properties and constraints, tables for mappings. The more structure you provide, the richer the graph.
2. Run doc_to_triples.py to extract the concept graph.
3. Run triple_polish.py in interactive mode. Answer the dialog questions to resolve ambiguities. Type 'done' when satisfied — remaining items go to the unresolved report.
4. Inspect the polished triple set. The concept graph IS the architecture — nodes are components, edges are relationships, violates triples are constraints.
5. Run triples_to_prompt.py with --mode task on the top-level token to generate the ordered implementation specification.

```
# Full text-to-architecture run
python3 doc_to_triples.py my_spec.docx --out-dir ./output
python3 triple_polish.py ./output/my_spec_triples.json
python3 triples_to_prompt.py ./output/my_spec_triples_polished.json \
    top_level_component --mode task
```

4.2 Prompt Building Mode

Generate precision prompts for any subsystem from a single polished triple set.

Once you have a unity-quality triple set, any token in the graph can become the target for a prompt. The graph walker pulls in all related context — dependencies, contracts, constraints, antipatterns, evidence — and assembles it into the requested format.

```
# See what tokens are available
python3 triples_to_prompt.py output/polished.json --list

# Generate a task spec for each pipeline phase
for phase in pipeline_phase_0 pipeline_phase_1 pipeline_phase_2 pipeline_phase_3; do
    python3 triples_to_prompt.py output/polished.json $phase \
        --mode task --out prompts/${phase}.txt
done

# Generate a full LLM system prompt for RM consciousness context
python3 triples_to_prompt.py output/polished.json resonant_mother --mode llm

# Batch: all tokens at once
python3 triples_to_prompt.py output/polished.json --all-tokens --mode task
# Output: prompts_task/ directory with one .txt per token
```

> **Reuse across sessions**
> The polished triple set persists between sessions. You only need to run doc_to_triples + triple_polish once per document. After that, triples_to_prompt can synthesize new prompts at any time for any token without re-extracting or re-polishing.

4.3 Progressive Module Architecture

Each specification document you process adds to the knowledge base. Run the pipeline on subsystem specs in sequence to build progressively richer concept graphs that capture the full architecture.

Recommended sequence

6. Start with the top-level white paper (architecture overview). This establishes the primary token vocabulary.
7. Process each phase specification as a separate document. Each phase doc produces its own polished triple set.
8. Process component specs for each phase. These inherit vocabulary from the parent triple set — use consistent token names to link them.
9. As each spec is processed, load the vocab pairs into RM's association memory (extend_rm_vocab.py — see Section 5).
10. Run triples_to_prompt on the leaf-level tokens to drive implementation sessions.

Level	Example document
L0 — Architecture	`RM_Architecture_WhitePaper.docx`
L1 — Phase specs	`Phase_0_Ingestion.docx, Phase_1_Domain.docx ...`
L2 — Component	`IntentVocabulary.docx, ArchetypeLibrary.docx ...`
L3 — Implementation	`GPUThrottleDaemon.docx, BootstrapV3.docx ...`

> **Progressive learning**
> Each document processed teaches RM the domain. The first document introduces vocabulary from scratch. Later documents extend the graph. By the tenth document, RM contributes structural observations not explicit in the source text.

4.4 Iterative Refinement Loop

The pipeline supports iterative refinement. Run the full loop, use the query battery to probe RM, then update the source document based on RM's responses and re-run.

```
# Round 1: extract, polish, load into RM
python3 doc_to_triples.py spec.docx --out-dir ./output
python3 triple_polish.py ./output/spec_triples.json
# extend_rm_vocab.py  (loads vocab into RM — see Section 5)
# run_rm_queries.py   (queries RM, collects responses)

# Round 2: update spec.docx based on RM responses, re-run
python3 doc_to_triples.py spec_v2.docx --out-dir ./output
python3 triple_polish.py ./output/spec_v2_triples.json
```

5. Integration with RM

The pipeline produces three artifact types that feed into RM's substrate. The integration scripts are next in the build queue.

Artifact	Script (pending)	What it does
vocab.json	`extend_rm_vocab.py`	Load Edinburgh association pairs into RM memory (port 8892)
triples_polished.json	`load_concept_graph.py`	Load triples as substrate training pairs
queries.json	`run_rm_queries.py`	Execute query battery against RM, collect feedback

> **Current state**
> The polished triple set (253 triples, unity achieved) is staged at /home/joe/sparky/doc_pipeline/output/RM_WhitePaper_triples_FINAL.json and ready for substrate load once the integration scripts are built.

6. File Reference

Path	Description
/home/joe/sparky/doc_pipeline/	`Pipeline root`
doc_to_triples.py	`Extractor script`
triple_polish.py	`Dialog polish loop`
triples_to_prompt.py	`Prompt synthesizer`
RM_Architecture_WhitePaper.docx	`Source document Rev 1.0`
output/ RM_WhitePaper_triples_FINAL.json	`253 clean triples — unity achieved`
output/ RM_WhitePaper_polish_report.json	`Full audit trail`
output/ RM_Architecture_WhitePaper_vocab.json	`1,275 Edinburgh association pairs`
output/ RM_Architecture_WhitePaper_queries.json	`23-item RM query battery`

7. Quick Reference

Minimal full run

```
cd /home/joe/sparky/doc_pipeline
python3 doc_to_triples.py <doc.docx> --out-dir ./output
python3 triple_polish.py ./output/<doc>_triples.json
python3 triples_to_prompt.py ./output/<doc>_triples_polished.json --list
python3 triples_to_prompt.py ./output/<doc>_triples_polished.json <token>
```

Common flags

Flag	Effect
--auto	triple_polish: skip dialog, auto repairs only
--mode task	triples_to_prompt: ordered step spec (default)
--mode llm	triples_to_prompt: full LLM system prompt
--mode rm	triples_to_prompt: terse RM substrate query
--list	triples_to_prompt: list all available tokens
--all-tokens	triples_to_prompt: batch generate for all top-level tokens
--depth N	triples_to_prompt: graph traversal depth (default 2)
--out FILE	triples_to_prompt: write prompt to file
--verbose	doc_to_triples: show extraction detail

Dialog answers (triple_polish interactive mode)

Input	Action
subject=token	Replace subject with token
object=token	Replace object with token
contract=token	Specify missing interface contract
polarity=negative	Re-encode anti-pattern as violates triple
discard	Remove triple entirely
skip	Leave unresolved — appears in report
done	Stop dialog early — remainder go to unresolved
help	Show format options for current failure mode