



Test-Harness: Struktur und Logging-Anforderungen

Das Ziel ist die **Kausalanalyse**: Wir müssen exakt wissen, welche Konfiguration (Input) zu welchem F1-Score (Output) geführt hat und *warum* (Logs & Zwischenergebnisse).

1. Vorgeschlagene Ordnerstruktur (Bestätigung deines Plans)

Du erstellst einen Haupt-Experiment-Ordner (z.B. `experiment_2025-11-06/`) und darin die Unterordner für jeden Test:

```
/experiment_2025-11-06/  
├── test_01_phase1_legend_only/  
├── test_02_baseline_monolith_all/  
├── test_03_baseline_swarm_only/  
├── test_04_baseline_specialist_chain/  
├── test_05a_chain_with_predictive_2d/  
├── test_05b_chain_with_polyline_2e/  
└── test_05c_chain_with_selfcorrect_3/
```

Jeder dieser Ordner muss die *identische* Art von Artefakten enthalten.

2. Was in JEDEN Test-Ordner gespeichert werden MUSS

Hier ist die Checkliste der Artefakte für *jeden* einzelnen Testlauf (z.B. für `test_04_baseline_specialist_chain/`):

A. Konfigurations-Snapshot (Die "Blaupause")

Das ist entscheidend für die spätere Nachvollziehbarkeit.

- **config_snapshot.yaml**: Eine exakte Kopie der `config.yaml`, die für *diesen* Lauf verwendet wurde.
 - *Warum?* Hier stehen die Model-Namen (`swarm_model`, `monolith_model`) und die Feature-Flags (z.B. `use_swarm_analysis: true`, `use_monolith_analysis: true`, `use_self_correction: false`).
- **prompts_snapshot.json**: Eine JSON-Datei, die alle Prompts (als Key-Value-Paare) speichert, die in diesem Lauf verwendet wurden (z.B. den genauen Text von `monolithic_analysis_prompt_template` etc.).
 - *Warum?* Der Prompt ist der wichtigste Parameter. Wenn du ihn änderst, musst du wissen, welcher Wortlaut zu welchem Ergebnis geführt hat.
- **test_metadata.md**: Eine einfache Textdatei, in die du die *Absicht* dieses Tests schreibst (z.B. "Test 4: Saubere Baseline der Spezialisten-Kette (Swarm -> GR -> Monolith-Connect-Only -> Fusion). Alle Helfer (2d, 2e, 3) sind DEAKTIVIERT.").
 - *Warum?* Damit du in zwei Wochen noch weißt, *warum* du diesen Test überhaupt

gemacht hast.

B. Pipeline- & LLM-Logs (Das "Protokoll")

Diese hast du bereits, sie müssen nur in den richtigen Ordner geleitet werden.

- **pipeline_...log**: Dein Haupt-Logfile.
 - *Warum?* Zeigt uns die Ausführungsreihenfolge, Timings und kritische Fehler (wie den Early stop in Phase 3).
- **llm_calls_...log**: Dein LLM-Logfile.
 - *Warum?* Das ist unser "Spion". Hier sehen wir die *exakte* Roh-JSON-Antwort des Modells (z.B. die 2 Verbindungen des Monolithen) und die finale Kritik.

C. Zwischenergebnisse (Die "Datenübergaben") - NEU & KRITISCH

Das ist der wichtigste Punkt, um die *Kausalkette* zu beweisen. Du musst deinen `pipeline_coordinator.py` so anpassen, dass er nach jeder relevanten Phase das Ergebnis als JSON in den Test-Ordner schreibt.

- **output_phase_2a_swarm.json**: Die reine Element-Liste, die der Swarm (Phase 2a) geliefert hat.
- **output_phase_2b_guardrails.json**: Die *bereinigte* Element-Liste, die die Guard Rails (Phase 2b) an den Monolithen übergeben haben.
 - *Warum?* Wir müssen den "Input" für den Monolithen sehen.
- **output_phase_2c_monolith.json**: Die reine Verbindungs-Liste, die der Monolith (Phase 2c) geliefert hat (z.B. die 2 Verbindungen aus dem ...092155-Log).
- **output_phase_2d_predictive.json** (falls aktiv): Der Graph *nachdem* Phase 2d lief.
 - *Warum?* Hier würden wir sehen, ob FT 11 -> FT 10 in *diesem* Schritt hinzugefügt wurde.
- **output_phase_2e_polyline.json** (falls aktiv): Der Graph *nachdem* Phase 2e lief.
 - *Warum?* Alternative zur obigen Prüfung.
- **output_phase_3_selfcorrect_ITER_1.json** (falls aktiv): Das Ergebnis nach der ersten Korrekturschleife.

D. Endergebnisse (Das "Resultat")

Alle finalen Artefakte, die deine Pipeline bereits erstellt.

- **results.json**: Das finale Analyse-Ergebnis.
- **kpis.json**: Die finalen KPIs (F1, Precision, Recall).
- **debug_map.png**: Die visuelle Ausgabe.
- **confidence_map.png**: Die Konfidenz-Karte.
- `cgm_data.json`, `report.html`, etc.

Anwendbare Implementierung

1. **Test-Skript anpassen:** Dein Test-Skript (z.B. scripts/run_test_with_validation.py) muss einen neuen Parameter annehmen: test_name (z.B. test_04_baseline_specialist_chain).
2. **Output-Pfad erstellen:** Das Skript erstellt den einzigartigen Ausgabeordner (z.B. experiment_2025-11-06/test_04_.../).
3. **Konfiguration speichern:** Das Skript muss *vor dem Start* die config.yaml und die Prompts in diesen Ordner kopieren (als "Snapshot").
4. **Pipeline anpassen:** Der PipelineCoordinator muss diesen einzigartigen test_output_dir erhalten und *alle* seine Logs und Ausgaben dorthin leiten.
5. **Zwischen-Logging (Wichtig):** Im pipeline_coordinator.py, am Ende von _run_phase_2a_swarm_analysis, _apply_guard_rails etc., fügst du Code hinzu, der das Ergebnis (z.B. swarm_result) als output_phase_2a_swarm.json in den test_output_dir speichert.

Wenn du das so umsetzt, hast du eine "Blackbox" für jeden Testlauf, die alle Inputs, Prozesse und Outputs enthält. Damit können wir die Halluzinationen (wie FT 11 -> FT 10) exakt der Phase zuordnen, die sie verursacht hat.



Teststrategie: "Pipeline Isolation & Integration"

Das Ziel ist, die Performance jeder Komponente (Phase) isoliert zu messen, bevor wir sie kombinieren. Jeder Testlauf sollte (falls möglich) gegen die "Ground Truth"-Daten von Einfaches P&I validiert werden, um `element_f1` und `connection_f1` zu erhalten.

Test 1: Baseline Phase 1 (Legenden-Erkennung)

- **Ziel:** Stabilität und Korrektheit der Legenden-Erkennung prüfen.
 - **Aktion:** Pipeline so konfigurieren, dass *nur Phase 1 (Pre-Analysis)* läuft.
 - **Deaktivieren:** Alle Phasen ab Phase 2.
 - **Datensammlung:**
 1. Prüfe die `legend_info.json` (für Einfaches P&I sollte sie leer sein, da keine Legende vorhanden ist).
 2. Prüfe mit einem Diagramm, das eine Legende *hat*. Wird die `symbol_map` korrekt befüllt?
 - **Erfolgskriterium:** Phase 1 läuft stabil und extrahiert Legenden-Daten korrekt, ohne bei fehlenden Legenden abzustürzen.
-

Test 2: Baseline "Simple P&ID" (Monolith "Alles-Finder")

- **Ziel:** Die Performance des "guten Laufs" (Monolith findet Elemente + Verbindungen) reproduzieren. Dies ist deine **Strategie für einfache Diagramme**.
 - **Aktion:** Pipeline-Logik (in `pipeline_coordinator.py`) so einstellen, dass sie dieser Kette folgt:
Phase 1 -> Phase 2c (Monolith) -> Phase 4 (Post-Processing)
 - **Wichtige Konfiguration:**
 1. **Monolith (Phase 2c):** Muss den Prompt für "Finde Elemente UND Verbindungen" verwenden (wie im alten ...205421-Lauf).
 2. **Deaktivieren:** Swarm (2a), Guard Rails (2b), Fusion (2c-Fusion), Predictive (2d), Polyline (2e), Self-Correction (3).
 - **Datensammlung:**
 1. Prüfe `results.json`: Wie hoch sind `element_f1` und `connection_f1`?
 2. Ist das Ergebnis sauber und frei von den Halluzinationen des ...092155-Laufs (z.B. keine FT 11 -> FT 10-Verbindung)?
 - **Erfolgskriterium:** F1-Scores sind hoch. Das Ergebnis ist dem "guten Lauf" (...205421) ebenbürtig.
-

Test 3: Baseline Phase 2a (Swarm "Elemente-Finder")

- **Ziel:** Die reine Performance des Swarm (Flash-Modell) bei der Element-Erkennung messen.
 - Aktion: Pipeline-Logik auf diese Kette setzen:
Phase 1 -> Phase 2a (Swarm) -> Phase 4 (Post-Processing)
 - **Wichtige Konfiguration:**
 1. **Deaktivieren:** Guard Rails (2b), Monolith (2c), Fusion (2c-Fusion), 2d, 2e, 3.
 - **Datensammlung:**
 1. Prüfe results.json: Wie hoch ist element_f1?
 2. Ist das connections-Array wie erwartet leer (oder fast leer)?
 - **Erfolgskriterium:** element_f1 ist hoch. Der Swarm liefert schnell und präzise *nur* Elemente.
-

Test 4: Baseline "Complex P&ID" (Spezialisten-Kette)

- **Ziel:** Die Performance deiner neuen Kern-Architektur (Swarm -> GR -> Monolith) *ohne* die fehlerhaften "Helfer"-Phasen messen. Dies ist deine **Strategie für komplexe Diagramme**.
 - Aktion: Pipeline-Logik auf deine designierte Kette setzen:
Phase 1 -> 2a (Swarm) -> 2b (Guard Rails) -> 2c (Monolith "Connect-Only") -> 2c (Fusion) -> Phase 4
 - **Wichtige Konfiguration:**
 1. **Monolith (Phase 2c):** Muss den Prompt für "Finde *nur* Verbindungen basierend auf dieser JSON-Liste" verwenden.
 2. **Deaktivieren:** Predictive (2d), Polyline (2e), Self-Correction (3).
 - **Datensammlung (Sehr wichtig):**
 1. Prüfe pipeline.log: Hat der Monolith (2c) die Element-Liste von 2b korrekt erhalten?
 2. Prüfe llm_calls.log: Was hat der Monolith (2c) *tatsächlich* geantwortet? (Im ...092155-Lauf waren es nur 2 Verbindungen).
 3. Prüfe results.json: Wie hoch ist connection_f1? Enthält es **Halluzinationen** (z.B. FT 11 -> FT 10)? *Es sollte nicht*, da 2d/2e deaktiviert sind.
 - **Erfolgskriterium:** Das Ergebnis ist **sauber** (keine Halluzinationen). Es darf unvollständig sein (niedriger connection_f1), aber es darf keinen Müll enthalten.
-

Phase 2: Debugging der "Helfer"-Phasen (Basierend auf Test 4)

Erst wenn Test 4 eine **saubere, aber unvollständige** Basis liefert, kannst du die "Helfer"-Phasen sinnvoll testen, um zu sehen, ob sie *helfen* oder *schaden*.

Test 5a: Isoliere Phase 2d (Predictive Completion)

- **Aktion:** Führe die Kette aus Test 4 aus, aber schalte **nur Phase 2d (Predictive)** hinzu.
- **Datensammlung:** Prüfe results.json. Hat sich der F1-Score *verbessert* (weil fehlende Verbindungen ergänzt wurden) oder *verschlechtert* (weil Halluzinationen wie FT 11 -> FT 10 hinzugefügt wurden)?
- **Ziel:** Kausalen Beweis finden, ob Phase 2d Rauschen hinzufügt.

Test 5b: Isoliere Phase 2e (Polyline Refinement)

- **Aktion:** Führe die Kette aus Test 4 aus, aber schalte **nur Phase 2e (Polyline)** hinzu.
- **Datensammlung:** Prüfe results.json. Hat diese CV-basierte Phase Verbindungen hinzugefügt oder entfernt? Hat sie den F1-Score verändert?
- **Ziel:** Kausalen Beweis finden, ob Phase 2e Rauschen hinzufügt.

Test 5c: Isoliere Phase 3 (Self-Correction)

- **Aktion:** Führe die Kette aus Test 4 aus, aber schalte **nur Phase 3 (Self-Correction)** hinzu.
- **Datensammlung:**
 1. Prüfe pipeline.log: Läuft der Loop überhaupt? (Im ...092155-Lauf stoppte er sofort wegen Quality Score (68.17) >= Min Score (60.0)).
 2. **Fix:** Du musst den Min Score (in der config.yaml) für den Early stop auf einen viel höheren Wert (z.B. 90.0) setzen, sonst wird er nie laufen.
- **Ziel:** Die Konfiguration von Phase 3 reparieren, damit sie bei echten Problemen überhaupt anspringt.

Ergebnis dieser Teststrategie

Nach diesen 5 Test-Kategorien hast du eine exakte Datenlage:

1. **Test 2** zeigt dir die (wahrscheinlich hohe) Baseline für *einfache* P&IDs.
2. **Test 4** zeigt dir die (wahrscheinlich unvollständige, aber saubere) Baseline für *komplexe* P&IDs.
3. **Test 5a/5b** beweist dir, welche der "Helfer"-Phasen die Halluzinationen (FT 11 -> FT 10) erzeugt hat, die deinen F1-Score im letzten Lauf zerstört haben.
4. **Test 5c** zeigt dir, wie du Phase 3 konfigurieren musst, damit sie funktioniert.

Mit diesen Daten kannst du dann die **Phase 0 (Complexity Analysis)** intelligent einstellen,

um je nach Diagramm zwischen Strategie (Test 2) und (Test 4 + reparierte Helfer) zu wechseln.