OrgFlow: End-to-End Organizational Process Discovery and Automation

Andrew Saffar [*1] **Joon Hwan Hong** [*2 3] **Marzia Nouri** [*1] **Sreenath Madathil** [*1 4]

Abstract

Organizational work is coordinated through informal communications — emails, messages, and calls — where process knowledge is embedded implicitly rather than formally documented, yet classical process mining requires structured event logs and recent LLM methods assume clean textual descriptions. We present OrgFlow, a three-agent pipeline that transforms raw multi-party transcripts into executable n8n workflows: a ReAct Process-Discovery Agent retrieves novel processes from a message store, a BPMN Generator (ProMoAI) converts each into BPMN via a POWL intermediate, and a Translation Agent compiles the BPMN into validated n8n JSON through sandboxed Python tool calls with typed diagnostic feedback driving retries. To enable rigorous evaluation, we synthesize a controlled benchmark of 3,080 messages across 243 scenarios grounded in the 55 PMo ground-truth BPMN models. On process discovery, our agent covers 42/55 ground-truth processes at Hungarian F1 0.754, outperforming K-Means and one-shot baselines; on the curated translation track, our Gemma-4-31B translator passes 32/55 pipeline models at mean F1 0.837, with retries substantially improving structural validity. End-to-end evaluation reveals process discovery as the dominant bottleneck, motivating future work on tighter integration between discovery and translation. Code and data are available at `https://github.com/MadathilSA/OrgFlow`.

[*]Equal contribution [1]School of Computer Science, McGill University, Montreal, Canada. [2]Quantitative Life Sciences, McGill University, Montreal, Canada. [3]Ludmer Centre for Neuroinformatics & Mental Health, Montreal, Canada. [4]Faculty of Dental Medicine and Oral Health Sciences, McGill University, Montreal, Canada.. Correspondence to: Any of the authors <>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Organizations carry out many recurring activities — processing orders, approving requests, handling customer support — through structured sequences of steps involving multiple people, systems, and decisions. *Business Process Management* (BPM) is the practice of analyzing, designing, executing, and automating these workflows; its standard representation is the *Business Process Model and Notation* (BPMN), a graphical notation that captures tasks, gateways (decision splits), and events as a directed flow. BPMN models enable conformance checking, optimization, and automation, but they have traditionally required either manual specification by domain experts or mining from structured event logs emitted by enterprise systems. Many organizational processes, however, are enacted through informal communication — email threads, messaging channels, meeting notes — that never reaches an enterprise event log.

Recent advances in large language models (LLMs) have made it plausible to bridge this gap. LLMs extract structured information from unstructured text with high accuracy, and several works have shown their utility for process modelling: (Kourani et al., 2025) reports up to 0.93 similarity when generating process models from clean textual descriptions; (Norouzifar et al., 2025) uses LLM-extracted declarative constraints to guide algorithmic discovery on real-world logs; and (Licardo et al., 2026) shows that a JSON intermediate representation substantially improves the reliability of LLM-generated process models. Each of these, however, operates *downstream* of our core problem: they assume clean textual descriptions or pre-existing event logs as input. The upstream transformation — from raw, noisy, multi-party organizational communication to a structured, executable process representation — remains unaddressed by LLM-based methods.

We propose **OrgFlow** is a three-agent LLM pipeline: a *Process-Discovery Agent* extracts business processes from communication corpus; a BPMN Generator converts these to BPMN via a POWL intermediate (ProMoAI); and a *Translation Agent* compiles BPMN into executable n8n workflows using tool calls into a sandboxed Python interpreter, with validation and detection of unsupported BPMN constructs. Workflows are deployed via the n8n REST API; a *Feedback Agent* captures execution outcomes and returns structured

diagnostics to the Translation Agent for retry.

Our investigation is organised around four threads:

1. **An end-to-end communication-to-workflow pipeline.** We build and evaluate a complete LLM system that ingests unstructured organizational communication and outputs deployed, trace-validated n8n workflows, linked by a structured feedback loop.

2. **A synthetic communication corpus with known ground truth.** We construct a data-generation pipeline (BPMN $\rightarrow$ path enumeration $\rightarrow$ multi-channel playout via LLM) that produces 3,080 messages across 243 scenarios grounded in the 55 PMo (Brissard et al., 2025) ground-truth models, enabling quantitative evaluation at every pipeline stage.

3. **BPMN-to-n8n LLM compilation with execution feedback.** We develop the Translation Agent and Feedback Agent described above; together they turn a BPMN graph into a deployable and executable n8n workflow, using deploy / activation / execution / logical outcomes as retry signal.

4. **Two evaluation tracks.** We report a *curated* track that isolates each stage on ground-truth inputs (Translation Agent on the 55 PMo ground-truth PMEs, Process-Discovery Agent on the 3,080-message corpus), and an *end-to-end* track that chains all stages. The curated track decouples translator quality from discovery quality. The end-to-end track exposes where the composed system actually breaks.

Research question. The central question of this paper is whether an iterative, tool-using LLM agent, paired with a downstream BPMN-to-n8n compiler driven by structured execution feedback, can recover and automate organizational business processes directly from raw multi-actor communications, without relying on structured event logs.

2. Related Work

Prior work on process discovery from email relies on pre-LLM clustering and NLP pipelines (Jlailaty et al., 2017; Elleuch et al., 2023), producing incomplete process fragments rather than deployable models. Recent work shows LLMs can generate POWL and BPMN from clean text (Kourani et al., 2025; 2024), with GRPO fine-tuning substantially reducing invalid outputs (Berti et al., 2025), and that JSON intermediates improve LLM-to-BPMN reliability over raw XML (Licardo et al., 2026)—findings we adopt directly. On workflow execution, BPMN-to-BPEL translation has been studied in the enterprise BPM literature (Ouyang et al., 2010), and recent LLM-driven work validates BPMN against interpreters such as SpiffWorkflow (Matei et al., 2026; Sekar et al., 2026) or compiles choreographies to smart contracts (Stiehle et al., 2025), but none targets consumer automation platforms (n8n, Zapier, Make) or closes the loop with stage-typed failure feedback. Our multi-agent design is informed by RLMs (Zhang et al., 2025) and CodeAct (Wang et al., 2024) for persistent sandbox execution, and contrasts with agentic task-graph systems such as FlowMind (Zeng et al., 2024) and Flow (Niu et al., 2025), which orchestrate LLM calls rather than translate formal process models into deployable automation JSON. A detailed discussion on related work is provided in the Appendix.

3. Models/Approach

3.1. Baselines

Hypothesis. Our central hypothesis for the process-discovery stage is that an iterative tool-using LLM agent, with explicit duplicate rejection and a novelty-driven sampler, recovers more distinct ground-truth processes from a multi-actor communication corpus than either (a) a non-LLM clustering pipeline that summarises message groups in isolation, or (b) a single-call LLM that sees the full corpus once.

K-Means clustering baseline. We embed each of the N=3,080 messages with sentence-transformers `all-MiniLM-L6-v2` and run K-Means with k=55, matching the number of ground-truth processes in the PMo dataset. For each cluster we send its messages to `gemini-2.5-flash` and ask for a single process description with the same schema as the agent's saves (name, description, ordered steps, actors, decision points). This baseline tests whether simple message clustering plus per-cluster LLM summarisation is enough.

One-shot LLM baseline. We place all 3,080 messages in a single prompt and ask `gemini-2.5-flash` to return a JSON array of distinct business processes, using the same schema as the agent's saves. The full prompt is in Appendix D. This uses the same model as our agent. It tests whether iterative tool use adds anything over a single LLM call.

Both baselines produce outputs with the same JSON schema as the agent, so the evaluation (Section 4.2.1) is identical across systems.

3.2. Our proposed model/approach

OrgFlow proposes an end-to-end pipeline, from process discovery to workflow automation. First, a ReAct agent discovers processes from raw communication. Then, from these processes, an n8n workflow is created by a translator

LLM agent, aided by an LLM enabled feedback loop. The workflow is then automatically deployed and executed using n8n REST API.

3.2.1. Process Discovery

The discovery stage is a ReAct agent (Yao et al., 2023): at each step the LLM emits a thought followed by one or more tool calls, receives structured observations, and decides the next move, until it either declares itself done or hits a step budget. Our implementation uses `gemini-2.5-flash` as the backbone, temperature 0, and a hard cap of 150 steps.

Message store and tools. The agent does not receive the raw corpus directly. It must explore it through tools. Messages are pre-embedded with `all-MiniLM-L6-v2`. The agent has access to eight tools, grouped by function:

- **Retrieval.** `search_messages(query)` does a cosine-similarity search over message embeddings; `filter_messages` takes structured fields (sender, receiver, channel, date range); `list_participants` returns every sender/receiver with message counts; `get_message_detail(id)` returns the full (un-truncated) text of a single message.
- **Save + self-inspection.** `save_discovered_process` persists a process (name, description, ordered steps, actors, decision points, evidence message IDs); `list_discovered_processes` returns the current saved inventory.
- **Coverage-driven exploration.** Two tools target recall directly. `find_uncovered_messages` returns an evenly-spaced sample of messages whose IDs are not cited as evidence by any saved process. `find_novel_messages` computes, for each message, the maximum cosine similarity to every saved-process embedding and returns those with the lowest maximum, messages semantically farthest from anything already saved. This is a direct pointer at process types the agent has not yet discovered, and was the single largest driver of the agent's per-process F1.

Duplicate rejection at save time. Each candidate save is embedded (name, description, and steps concatenated) and compared to every existing saved-process embedding. If cosine similarity is at least 0.85 to any existing save, the save is rejected with a message that names the duplicate, so the agent can pivot to a different process. After every ten successful saves, the tool response appends the current saved inventory as a coverage reminder. This rejection step was essential for controlling near-duplicate saves once we strengthened the prompt against early termination (see ablation in Section 6).

Stopping criteria. The system prompt (Appendix D) explicitly forbids termination until the agent has (a) saved a substantial inventory, (b) called `list_discovered_processes` and reviewed what it has, (c) called `find_novel_messages` at least three times with the most recent call producing only near-matches of existing saves, and (d) verified via targeted probes that the top novel/uncovered messages do not support any new coherent workflow. Stock ReAct would stop much earlier and cover far fewer ground-truth processes. This check plus the novelty tool more than double coverage on our corpus (Section 6).

Output. The agent produces a directory of JSON files, one per discovered process, with the save schema described above. No BPMN model is emitted at this stage. The BPMN Generator (ProMoAI, Section 3.2.2) converts each of these natural-language process descriptions into BPMN XML before the Translation Agent consumes them.

3.2.2. Workflow Automation

Input and construct mapping. The stage takes a *Process Model Elements* (PME) JSON from the PMo dataset (Brissard et al., 2025) — a flat object with six lists (`tasks`, `events`, `gateways`, `pools`, `sequenceFlows`, `messageFlows`) that carries all control flow, gateway conditions, swimlane pools, and cross-pool message flows. When the upstream input is BPMN XML (end-to-end track) we first convert it to PME deterministically. The Translation Agent uses a nine-node n8n allowlist covering all 1,276 elements of the corpus (Figure 10 in Appendix G): `Task`/`Manual` → `set`; `Service`/`Send`/`User` → `code`; `Receive` and `IntermediateTimerEvent` → `wait`; `StartNoneEvent` → `manualTrigger`; message events → `webhook`; `EndNoneEvent` → `noOp`. Gateways decompose by fan-out/fan-in: XOR splits use `if` (fan-out $= 2$) or `switch` (fan-out > 2); AND splits use a `noOp` node; joins with fan-in ≥ 2 use `merge` only when feeders are not downstream of an XOR split (otherwise the merge deadlocks. Multi-gateway routing uses a single `_run_index` field plus per-gateway stride arithmetic for deterministic simulation.

Translation Agent. The Translation Agent is an LLM with a single tool, `submit_workflow`, whose JSON schema derives from the `WorkflowCreate` Pydantic model (Appendix G). In *code_interpreter* mode, the LLM does not write n8n JSON directly; it emits a Python program into a sandboxed Jupyter kernel holding a small helper library: `add_node`, `add_edge`, `add_if_node`, `add_switch_node`, `add_decider`, `add_loop_counter`, and `build_workflow`. The helpers assemble the n8n JSON internally.

Validation (static pre-check and post-hoc). Before the LLM is invoked, a static rule `detect_n8n_unrepresentable` flags constructs n8n cannot express (multi-pool BPMN, cross-pool message flows, event-based gateways, multiple start-message events) and classifies such runs as `failure_source = n8n_limitation`, separating LLM failures from platform limits. The LLM's output is then passed through `validate.py`: a Pydantic schema (unique node names, every connection references an existing node, graph connected to a trigger) plus four procedural checks — `validate_loop_termination`, `validate_branch_independence`, `validate_merge_inputs` (merge-deadlock check), and `validate_gateway_conditions`. A structural failure raises a typed error whose message becomes the next-attempt user turn.

Retry loop and Feedback Agent. Each model is run for up to $K=3$ attempts (initial plus two retries; pass@3). After a failed attempt, the full prior turn (reasoning, tool call, validator error) is appended to the conversation so attempt 3 sees the complete failure trajectory. We interpose a Feedback Agent that converts each *(workflow JSON, engine response)* pair into a typed JSON diagnostic with `failure_stage` $\in$ {`deploy`, `activation`, `execution`, `logical`, `none`}, `failure_source` $\in$ {`translation_error`, `bpmn_quality`, `n8n_limitation`, `unknown`}, plus `bpmn_construct_responsible` and `suggested_fix` (full schema in Appendix G). The diagnostic is the input to the next retry. The translator reads `bpmn_construct_responsible` and `suggested_fix`. This typed output both shapes self-correction and provides a uniform ontology for reporting failure modes in Section 6.4.

Deploy and execute. Workflows are POSTed to the n8n REST API, activated, and executed against a BPMN-derived trace (a path through the PME's sequence flows, injected through the trigger with explicit `_run_index`). The execution trace — nodes that actually fired — is compared against expected BPMN tasks using the trace-replay F1 of Section 4.2.2. Activation or execution failures feed back into the retry loop; partial logical success with F1 below threshold is recorded as a partial pass.

4. Dataset and evaluation metrics

4.1. PMo Dataset to Communication Corpus

The PMo Dataset (Brissard et al., 2025) is a curated benchmark of 55 expert-validated process model and textual description pairs. This dataset was specifically designed for LLM-based process modeling research, with representation choices motivated by token efficiency and schema-following capabilities. However, evaluating process discovery from organizational communications requires a corpus in which the ground-truth process structure underlying each conversation is precisely known, a condition that real-world communication archives cannot satisfy due to the absence of verified process annotations. We therefore synthesize a corpus of multi-channel communication transcripts directly from the PMo Dataset, using its BPMN models as ground truth and generating realistic email, SMS, and call transcript exchanges that enact each process's execution paths. We generated 243 multi-channel transcripts (3,080 messages across email, SMS, and call channels) directly from the 55 BPMN models in the PMo Dataset. Execution paths are enumerated via depth-first search over sequence flows, with XOR gateways producing distinct variants (capped at 32 paths per process via stratified sampling), and tasks are assigned to channels by a deterministic keyword rule engine. Transcripts are generated using Claude Sonnet 4.6 with structured prompts carrying nine generation rules, few-shot examples, and per-scenario task and actor specifications, and are validated against automated checks for field presence, timestamp ordering, and task coverage; all 243 transcripts achieved a task coverage ratio of 1.0. For evaluation, BPMN annotations and metadata are stripped, structured IDs are replaced with opaque SHA1 hashes, and scenarios are anchored to random offsets within a compressed three-month window so that messages from all 243 scenarios interleave chronologically, preventing the discovery agent from recovering scenario boundaries through temporal or structural cues. Further details are provided in the Appendix B.2.

4.2. Evaluation Metrics

4.2.1. Process Discovery

At the Process-Discovery Agent's output, a discovered process is a natural-language description (name, description, ordered steps, actors, decision points) and each ground-truth process is a free-text description; we therefore evaluate at the description level via **semantic matching** between the two sides. After the BPMN Generator converts discovered descriptions to BPMN, a complementary **structural comparison** against the PMo ground-truth BPMN is possible: we compute *Task F1* (fuzzy-matched task names), *Gateway F1* (gateway-type counts), and *Flow F1* (sequence-flow pairs), plus an LLM-as-judge similarity score. These end-to-end numbers are reported in Section 6.4 and the end-to-end results section.

Embedding and matching. Each discovered process is embedded by concatenating NAME $\oplus$ DESCRIPTION $\oplus$ STEPS and encoding with sentence-transformers `all-MiniLM-L6-v2`. The 55 GT descriptions are encoded as-is. For every (discovered, GT) pair we compute cosine similarity on L2-normalised vectors. Given a thresh-

old τ, the pair is called a *match* if similarity $\geq \tau$. Our default threshold is τ=0.5; we also report a threshold sweep over $[0.35, 0.60]$ to show that system rankings are stable (Section E).

Three matching schemes. We report F1 under three schemes because each captures a different kind of error:
Lenient F1: every discovered process is assigned to its best GT above τ, with many-to-one allowed. True positives = matched discoveries; false positives = unmatched discoveries; false negatives = GTs never matched by any discovery. Rewards coverage but is invariant to over-saving the same GT.
Strict F1: every GT is counted at most once across matches. Over-saves of the same GT become false positives, so strict F1 penalises duplicate discoveries directly.
Hungarian F1: one-to-one optimal assignment that maximises total similarity (solved with the Hungarian algorithm), followed by thresholding at τ. This is the strictest scheme and the most informative about whether the system produces a clean, non-redundant model inventory.

Reporting. We run the ReAct agent three times with different random seeds and report the median and the min/max range across runs, because `gemini-2.5-flash` exhibits non-trivial run-to-run variance at temperature 0 (see Section 6.2).
Domain-level metric. As a complementary view, we also report a **domain-level F1**. We prompt `gemini-2.5-pro` once to cluster the 55 ground-truth descriptions into twelve domains (e.g. *retail and orders*, *hr and employment*, *manufacturing and qc*). A domain is *covered* if any of its member GTs is matched by a discovered process (using the same τ=0.5 threshold), and we report the analogous greedy-F1 and Hungarian-F1 numbers over the 12 domains. Domain-level metrics are less discriminating than per-process F1 but directly answer the question "did the system miss an entire subject area?"

4.2.2. Workflow Automation

We report four evaluation axes for the workflow-automation stage, corresponding to the four pipeline failure points at which a model can drop out.
Structural pass@k. For each model, we allow the Translation Agent up to k attempts (initial attempt plus $k-1$ retries); a model is counted as *structurally passing at* k if any of the first k attempts produces a workflow that clears `validate_workflow_create` (the Pydantic schema plus the four custom checks in Section 3.2.2). We report pass@1, pass@2, and pass@3 (the paper uses k=3); pass@1 is the zero-feedback baseline, pass@3 isolates the cumulative contribution of the Feedback Agent.

Deployment stage classification. Each structurally valid workflow is POSTed to n8n and traced through four lifecycle stages, each a potential drop-out point: `deploy` (workflow JSON accepted by the n8n API), `activation` (workflow enters the active state with a trigger), `execution` (runtime completes without thrown errors), and `logical` (execution produces the expected output). Each failing model is labelled with the earliest stage at which it failed, giving a strict hierarchy: `deploy` dominates `activation` dominates `execution` dominates `logical`. A model is a *full pipeline pass* only if every stage returns `none`.
Trace-replay F1. For every workflow that reaches the execution stage we compute an F1 between the set of n8n nodes that fired and the set of BPMN tasks the injected trace was expected to cover. Let E be the multiset of executed n8n nodes and G the BPMN ground-truth task set for the trace; precision is $|E \cap G|/|E|$ and recall is $|E \cap G|/|G|$, with the intersection computed on the PME-ID labels that the Translation Agent is required to preserve as node names. F1 is the harmonic mean. We report mean F1 both over all scored models and restricted to pipeline-pass models (to separate the "how good is a passing workflow" question from the "how often do we pass" question).
Pre-emptive `n8n_limitation` classification. Models for which `detect_n8n_unrepresentable` fires (Section 3.2.2) are counted as failures with `failure_source = n8n_limitation`, even if the Translation Agent would subsequently have succeeded. This is deliberately conservative: it attributes to the pipeline a failure that reflects a genuine expressiveness gap between BPMN and n8n (multi-pool, event-based-gateway, multiple start-message events), rather than treating such models as out-of-scope.
Reporting. For each run we report the full per-model row (model id, element / task / gateway counts, `regen_success`, `regen_pass_at`, `pipeline_status`, F1, `failure_stage`, `failure_source`, missing nodes) as a CSV artefact. Appendix H documents the exact schema.

5. Experimental details

The Process-Discovery Agent uses `gemini-2.5-flash` via the Google GenAI API. For the curated workflow-automation track reported in Section 6.4, the Translation Agent and Feedback Agent both use `gemma-4-31B-it` via FPT AI Factory (`mkp-api.fptcloud.com`). Corpus transcripts were generated with Claude Sonnet 4.6 via the Anthropic API (Section **??**). The end-to-end track uses a different model configuration, documented alongside its results.
Process discovery. The ReAct agent is configured with temperature 0 and a maximum of 150 steps. We run it three

times with different random seeds ($\{42, 1, 7\}$) to characterise variance. Each run is evaluated against the 55 PMo ground-truth descriptions (Section 4.2.1). We also record token usage and wall-clock time per run (Section E).

Workflow automation. For both the translator and feedback agents, we allow an initial attempt + 2 retries. This lets the system improve based on feedback while limiting inference cost. We parallelize PME-to-n8n translation using 4 workers. Experiments were also run in isolation (e.g. translating ground-truth PMEs to n8n) using the same models and hyperparameters. The OrgFlow end-to-end pipeline was run 3 times. All prompts are in Appendix D.

6. Results & Discussion

We report results along the two evaluation tracks defined in Section 3.2. Section 6.1 covers the Process-Discovery Agent against the K-Means and one-shot LLM baselines on the 3,080-message corpus. Section 6.4 reports the Translation Agent and Feedback Loop in isolation on the 55 PMo ground-truth PMEs under Gemma-4-31B. End-to-end numbers, which chain all stages and expose the composition of discovery and translation error, are reported in the end-to-end track subsection.

6.1. Process discovery: headline results

Table 1 reports F1 under each matching scheme at threshold τ=0.5, for the two baselines and the three independent ReAct agent runs. The best ReAct run (`seed=7`) discovers 67 processes and covers 42 of the 55 ground-truth processes under lenient matching. The median ReAct run covers 37.

We make three observations. First, the K-Means baseline is weak (Hungarian F1 0.364): cluster centroids do not respect process boundaries, and many clusters either fragment a single process or mix several. Second, the one-shot LLM baseline is strong (lenient F1 0.745, Hungarian F1 0.716): a single call with the full corpus already produces reasonable process descriptions. Third, the ReAct agent beats the one-shot on every per-process metric at the median — lenient F1 0.810, Hungarian F1 0.754, and 37 of 55 ground-truth processes covered vs. 31 for the one-shot. The best single run (seed 7) covers 42 of 55.

At the domain level, the agent's advantage is in coverage: seeds 42 and 7 cover all 12 ground-truth domains, while both baselines miss 2 and the worst agent seed (1) misses 1. Domain-greedy F1 is essentially tied across the agent median (0.953) and the one-shot (0.950). On domain-Hungarian F1 the one-shot is slightly ahead (0.895 vs. 0.852), because the agent's higher number of saves reduces its domain-level precision.

Individual agent runs vary substantially (Hungarian F1 from 0.541 to 0.755 across three seeds). We return to this in Section 6.2.

Reaching this configuration required four targeted iterations adding one component at a time. The marginal contribution of each change is reported in Appendix E.

6.2. Variance and limitations

Our three ReAct runs (`gemini-2.5-flash`, temperature 0, max 150 steps) produced strikingly different results: Hungarian F1 ranged from 0.541 (seed 1) to 0.755 (seed 42), and the number of discovered processes ranged from 30 to 67. Agent trajectories differed substantially across seeds: in the best run the agent invoked `find_novel_messages` repeatedly and followed each call with targeted searches, while in the worst run it never called the novelty tool and instead looped on coverage-style sampling.

This is the principal limitation of the reported results. Two caveats follow. First, the headline numbers in Table 1 should be read as the median and range across three seeds, not as a point estimate of the agent's true expected performance. Second, a comparison to the one-shot baseline at the median would benefit from many more samples than we ran. The ordering we report (agent median beats baselines) is robust on coverage-oriented metrics (lenient F1, covGT) but is closer to a tie on precision-oriented metrics (strict, Hungarian) and could flip under a different three-seed sample. Making the novelty tool effectively mandatory, by strengthening the stopping criterion to require several `find_novel_messages` calls before termination, is a natural direction for reducing run-to-run variance. We leave it to future work.

6.3. Error analysis

Thirteen GT processes (ids `21, 22, 23, 24, 25, 35, 36, 37, 41, 46, 51, 53, 54`) remain uncovered even by the best ReAct run. Inspecting their messages suggests three recurring failure modes: (i) *rare processes* supported by very few messages, where semantic search does not surface them until much of the evidence has already been claimed by adjacent processes; (ii) *near-variants* of already-saved processes (e.g. a sub-type of an order-fulfilment workflow) that the dedup check at 0.85 rejects; and (iii) *processes with overlapping actors*, where the agent assigns messages to the first process it finds rather than splitting out the second. We leave addressing these to future work.

6.4. Workflow Automation

We evaluate the Translation Agent and Feedback Loop in isolation on the 55 PMo ground-truth PMEs, with `gemma-4-31B-it` on both agents and pass@3 (initial

Table 1. Process discovery results at similarity threshold τ=0.5. n is the number of discovered processes; *covGT* is the count of the 55 ground-truth processes covered under lenient matching; *covDom* is the count of the 12 domains covered (see Section 4.2.1). Best result in each column is **bold**.

The mean±std summary across the three seeds is reported in the bottom row.

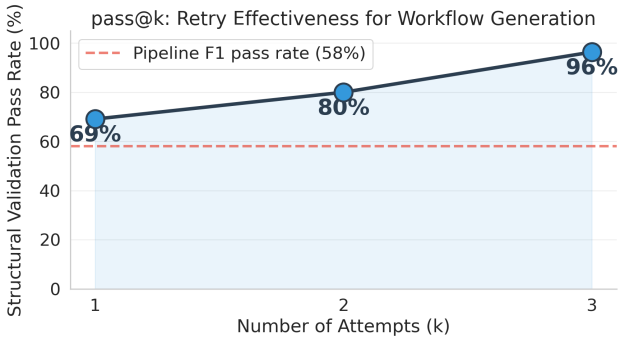
System	n	covGT	covDom	Len. F1	Strict F1	Hung. F1	Dom-G F1	Dom-H F1
K-Means baseline (k=55)	55	20	10/12	0.537	0.364	0.364	0.719	0.519
One-shot Flash (single call)	40	31	10/12	0.745	0.653	0.716	0.950	**0.895**
ReAct agent, seed 1	30	20	11/12	0.571	0.471	0.541	0.912	0.852
ReAct agent, seed 42	51	37	**12/12**	0.810	**0.698**	**0.755**	**0.959**	0.879
ReAct agent, seed 7	**67**	**42**	**12/12**	**0.865**	0.689	0.754	0.953	0.814
ReAct median	51	37	**12/12**	**0.810**	0.689	0.754	0.953	0.852
ReAct mean ± std	49 ± 19	33 ± 12	$11.7/12$	0.75 ± 0.16	0.62 ± 0.13	0.68 ± 0.12	0.94 ± 0.03	0.85 ± 0.03

attempt + two retries).

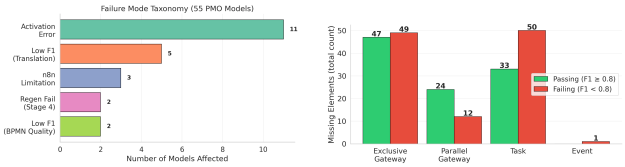
Headline and retry breakdown. The translator reaches **53/55 (96.4%) structural pass@3**; of those, **32/55 (58.2%) also pass the full pipeline** (deployed workflow activates, executes, trace-replay F1 ≥ 0.8). Mean trace-replay F1 is **0.837** across the 39 models that produced an executing workflow, and **0.888** restricted to pipeline-pass models. Retries account for most of the coverage: 38/55 (69%) at attempt 1, +6 at attempt 2 (pass@2 = 80%), +9 at attempt 3 (pass@3 = 96.4%), Figure 1a. The only persistent Stage-4 failures, pmo-17 and pmo-18, are the two largest models in the corpus (16 and 22 gateways).

What fails, and why. Figure 1b summarises failure modes among models with a structural mismatch. By `failure_source`: **translation_error** 30, **bpmn_quality** 17, **n8n_limitation** 3 (the latter flagged pre-emptively by `detect_n8n_unrepresentable`). Among failures that passed structural validation but failed downstream, 14/21 fail at n8n *activation* and 7/21 at *logical* execution. Figure 1c decomposes the dropped elements: exclusive and parallel gateways dominate both pass and fail populations, but pass-side misses are single-gateway slips in otherwise-correct workflows, whereas fail-side misses compound with missing tasks.

Complexity drives failure; trivial models pass. The passing and failing populations separate cleanly on complexity (Figure 1dd): pipeline-pass models concentrate below ∼20 elements and ∼6 gateways, while the failing tail extends to 50 elements and 22 gateways, with gateway count the sharper discriminator. Once the agent gets the gateway-handling pattern right, the trace matches nearly exactly; once it errors on a gateway join, the downstream subgraph drops out and F1 collapses. The simplest models (e.g. pmo-54: 5 elements, 0 gateways, pass@1 in 30 s and 10,953 tokens) pass at attempt 1: roughly 69% of the corpus follows this trivial pattern on first try.

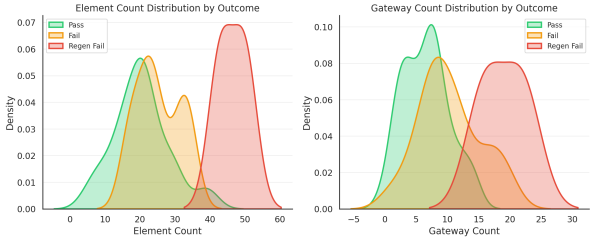


(a) Structural pass rate by attempt. Retries close most of the gap from initial pass to near-complete coverage.



(b) Failure-mode counts.

(c) Missing elements by type.



(d) Element- and gateway-count KDEs split by outcome.

Figure 1. Workflow-automation results on the 55-model Gemma-4 curated track. (a) retries account for most of the gap from initial pass (69%) to pass@3 (96%). (b) classification of failures by source and stage. (c) missing-node counts bucketed by BPMN element type, split by pipeline outcome. (d) shows complexity drivers on the workflow-automation only experiment.

Cascade example: feedback at work. Figure 6 shows the retry cascade on pmo-17. Attempt 1 fails with a Python brace-balance error when the agent hand-writes the `conditions` dict for an XOR gateway. The validator

points to the pre-loaded `add_if_node` helper. Attempt 2 clears the syntax issue but triggers a *merge deadlock*: the agent uses an n8n `merge` for a parallel-gateway join whose feeders are downstream of an XOR split, so one branch never fires and the merge blocks. The Feedback Agent names the exact construct (`ParallelGateway_4`), flags `ParallelGateway_3` and `ParallelGateway_4` as dangerous AND joins, and prescribes specific surgery: delete the merges and wire each feeder directly to the successor. Attempt 3 applies the fix and passes.

Takeaways. (1) *Translator ceiling is high on clean input*: 96.4% structural pass@3. (2) *Gateway handling dominates failures*: almost every failure is a gateway-syntax slip (hand-written `conditions`) or a merge-topology error (parallel merge on XOR-dependent feeders). (3) *The feedback loop closes the pass@1–pass@3 gap* (69%→96.4%) via structured per-failure guidance rather than additional model capacity.

6.5. End-to-End Track Results

Table 2 shows very low scores on BPMN Generation: LLM-as-a-judge ranks the average similarity between the generated BPMN and the ground-truth near 0.1 for all of our end-to-end runs. This semantic approach appears to be much stricter than the structural evaluation, which finds mean overall F1 around 0.35. It seems that the main weak point is the task name matching: we find extremely low scores for task F1. This suggests that the generated task names do not match the ground truth, which might lead the LLM to not recognize the patterns from the original BPMNs.

We also find that there is a large difference in discovery quality between runs. The most successful run in that aspect covers 13 more processes on average than the two other runs, and achieves significantly higher Hungarian F1.

Translation and execution are remarkably effective, even from generated BPMNs. We note that the run with the better discovered processes leads to the highest pass rate: it reaches almost 75%.

Fig. 2 shows that the first iteration of the feedback loop yields the biggest improvement. The second feedback iteration gives little to no improvement, indicating that errors that could not be overcome within the first feedback cycle are deeper rooted, and may need many more feedback iterations or even not be solvable by this system at all. Thus, we find that for our purposes, one iteration of the feedback loop is sufficient.

7. Conclusion

We presented OrgFlow, an end-to-end pipeline that transforms raw multi-actor organizational communications into deployable n8n workflows.

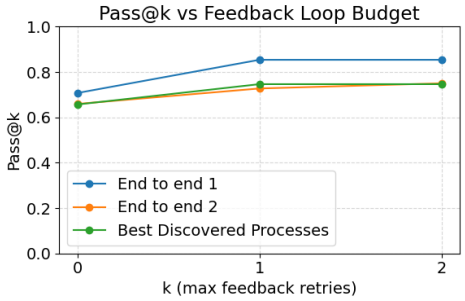


Figure 2. Feedback Loop Pass@k for each end-to-end run

Table 2. Comparison of results between end-to-end runs, end-to-end run starting at the best iteration of process discovery, and curated results from ground truth models.

Stage	Metric	E2E (avg. of 2 runs)	E2E (best discovery)	Curated
Discovery	Hungarian F1	0.64	0.75	n/a
Discovery	GT processes covered	29/55	42/55	n/a
BPMN Generation	Task F1 vs GT	0.04	0.03	n/a
BPMN Generation	Overall F1 vs GT	0.35	0.34	n/a
BPMN Generation	LLM judge similarity	0.11	0.14	n/a
Translation	Structural pass@3	92.7%	100%	96.4%
Execution	Pipeline pass rate	68.2%	74.6%	60.4%
Execution	Mean trace F1	0.89	0.89	0.84

On process discovery, our best run covers 42/55 ground-truth processes at Hungarian F1 0.754, and the three-seed median outperforms both K-Means and one-shot LLM baselines across all per-process metrics. On workflow automation, the Gemma-4 translator with two feedback-driven retries achieves 32/55 pipeline passes with a mean trace-replay F1 of 0.837. End-to-end evaluation identifies process discovery—not translation—as the primary bottleneck.

Two key limitations motivate future work. First, substantial run-to-run variance in the discovery stage highlights a robustness gap. Second, the realism of the synthetic corpus remains unvalidated, raising questions about external generalization. Addressing these issues—by improving discovery stability, evaluating on real-world communication corpora (e.g., Enron, Avocado), and systematically studying noise sensitivity—forms the next step toward practical deployment.

8. Contributions

Andrew Saffar Worked on the workflow automation section (in particular the feedback loop) and the end to end pipeline. **Joon Hwan Hong** Workflow automation section: Agent tooling (JSON tooling and Agent Jupyter environment setup), PME to n8n JSON agent. Visualiza-

tions/figures. **Marzia Nouri** Led the process-discovery stage, including the design, implementation, and analysis of the proposed methods. **Sreenath Madathil** Led the synthetic communication corpus creation from PMo Dataset.

All authors contributed to writing and editing of the manuscript.

9. Acknowledgments

We thank Dr. Siva Reddy for his guidance and mentorship, and Dr. Prudhvi Vatala for proposing this project.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Back, C. O., Slaats, T., Hildebrandt, T. T., and Marquard, M. DisCoveR: Accurate and efficient discovery of declarative process models. *International Journal on Software Tools for Technology Transfer*, 24(4):563–587, 2022. doi: 10.1007/s10009-021-00616-0.

Berti, A., Wang, X., Kourani, H., and van der Aalst, W. M. P. Specializing large language models for process modeling via reinforcement learning with verifiable and universal rewards. *Process Science*, 2:26, 2025. doi: 10.1007/s44311-025-00034-4.

Brissard, A., Cuppens, F., and Zouaq, A. What is the best process model representation? A comparative analysis for process modeling with large language models. In *Proceedings of the AI4BPM Workshop at BPM 2025*, 2025. doi: 10.5281/zenodo.15857589.

Busch, K. and Leopold, H. Towards a benchmark for large language models for business process management tasks, 2024.

Elleuch, M., Alaoui Ismaili, O., Laga, N., and Gaaloul, W. Process fragments discovery from emails: Functional, data and behavioral perspectives discovery. *Information Systems*, 118:102229, 2023. doi: 10.1016/j.is.2023.102229.

Georgakopoulos, D., Hornick, M., and Sheth, A. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995. ISSN 1573-7578. doi: 10.1007/BF01277643. URL https://doi.org/10.1007/BF01277643.

Jlailaty, D., Grigori, D., and Belhajjame, K. Business process instances discovery from email logs. In *Proceedings of the 2017 IEEE International Conference on Services Computing (SCC)*, pp. 19–26. IEEE, 2017. doi: 10.1109/SCC.2017.12.

Klievtsova, N., Benzin, J.-V., Mangler, J., Kampik, T., and Rinderle-Ma, S. Process modeler vs. chatbot: Is generative AI taking over process modeling? In *Process Mining Workshops. ICPM 2024*, volume 533 of *Lecture Notes in Business Information Processing*. Springer, Cham, 2025. doi: 10.1007/978-3-031-82225-4_47.

Klimt, B. and Yang, Y. The Enron corpus: A new dataset for email classification research. In Boulicaut, J.-F., Esposito, F., Giannotti, F., and Pedreschi, D. (eds.), *Machine Learning: ECML 2004*, volume 3201 of *Lecture Notes in Computer Science*, pp. 217–226. Springer, Berlin, Heidelberg, 2004. doi: 10.1007/978-3-540-30115-8_22.

Kourani, H., Berti, A., Schuster, D., and van der Aalst, W. M. P. Promoai: Process modeling with generative ai. 2024. doi: 10.48550/ARXIV.2403.04327. URL https://arxiv.org/abs/2403.04327.

Kourani, H., Berti, A., Schuster, D., and van der Aalst, W. M. P. Evaluating LLMs on business process modeling: Framework, benchmark, and self-improvement analysis. *Software and Systems Modeling*, 2025. doi: 10.1007/s10270-025-01318-w.

Laga, N., Elleuch, M., Gaaloul, W., and Alaoui Ismaili, O. Emails analysis for business process discovery. In *Proceedings of the Workshop on Algorithms and Theories for the Analysis of Event Data (ATAED 2019)*, volume 2371 of *CEUR Workshop Proceedings*, pp. 54–70, 2019. URL https://ceur-ws.org/Vol-2371/ATAED2019-54-70.pdf.

Leymann, F. Bpel vs. bpmn 2.0: Should you care? In *Business Process Modeling Notation*, volume 67 of *Lecture Notes in Business Information Processing*, pp. 8–13. Springer, Berlin, Heidelberg, 2010. doi: 10.1007/978-3-642-16298-5_2.

Li, X., Ni, L., Li, R., Liu, J., and Zhang, M. MaD: A dataset for interview-based BPM in business process management. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, June 2023. doi: 10.1109/IJCNN54540.2023.10191898.

Licardo, J. T., Tanković, N., and Etinger, D. BPMN assistant: An LLM-based approach to business process modeling. *Applied Sciences*, 16(5):2213, 2026. doi: 10.3390/app16052213.

Mangler, J. and Klievtsova, N. Dataset: Textual process descriptions and corresponding BPMN models, March 2023. URL https://doi.org/10.5281/zenodo.7783492.

Matei, I., Zhenirovskyy, M., Sekar, P. K. M., and Wong, H. Y. Automated BPMN model generation from textual process descriptions: A multi-stage LLM-driven approach. *arXiv preprint arXiv:2604.12105*, 2026. URL https://arxiv.org/abs/2604.12105.

Niu, B., Song, Y., Lian, K., Shen, Y., Yao, Y., Zhang, K., and Liu, T. Flow: Modularized agentic workflow automation. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2501.07834.

Norouzifar, A., Kourani, H., Dees, M., and van der Aalst, W. Integrating domain knowledge into process discovery using large language models, 2025.

Oard, D., Webber, W., Kirsch, D., and Golitsynskiy, S. Avocado research email collection, 2015. URL https://catalog.ldc.upenn.edu/LDC2015T03. LDC2015T03.

Ouyang, C., Dumas, M., ter Hofstede, A. H. M., and van der Aalst, W. M. P. Pattern-based translation of bpmn process models to bpel web services. In *Web Services Research for Emerging Applications: Discoveries and Trends*, pp. 545–566. IGI Global, 2010. doi: 10.4018/978-1-61520-684-1.ch023.

Sekar, P. K. M., Matei, I., Zhenirovskyy, M., Wong, H. Y., Kohmura, S., Hotta, S., and Inomata, A. Automatic generation of executable BPMN models from medical guidelines. *arXiv preprint arXiv:2604.07817*, 2026. URL https://arxiv.org/abs/2604.07817.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Stiehle, F., Weytjens, H., and Weber, I. On LLM-assisted generation of smart contracts from business processes. *arXiv preprint arXiv:2507.23087*, 2025. URL https://arxiv.org/abs/2507.23087.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zeng, Z., Watson, W., Cho, N., Rahimi, S., Reynolds, S., Balch, T., and Veloso, M. FlowMind: Automatic workflow generation with LLMs. In *Proceedings of the 4th ACM International Conference on AI in Finance (ICAIF)*, 2024. URL https://arxiv.org/abs/2404.13050.

Zhang, A. L., Kraska, T., and Khattab, O. Recursive language models, 2025.

A. Related Work

We organize related work along three aspects of our problem: (1) process discovery from unstructured communications, (2) workflow automation, and (3) multi-agent LLM architectures for complex analytical tasks.

A.1. Process Discovery

The challenge of discovering business processes from email data has been studied since the late 2000s, but prior work relies on pre-LLM NLP techniques. (Jlailaty et al., 2017) proposed a two-step hierarchical clustering approach: Word2Vec topic clustering groups emails by process type, then instance sub-clustering uses content similarity, named entries, and participant overlap. (Elleuch et al., 2023) extended this to multi-perspective discovery (functional, data, behavioral, organizational) using speech act detection for activity recognition, producing *process fragments* from the Enron corpus. A consistent finding across this literature is that due to the unstructured nature of email logs data, traditional process mining techniques could not be applied or at least not directly applied (Elleuch et al., 2023). All approaches require substantial preprocessing pipelines and produces incomplete process representations. No prior work uses LLMs for the extraction step, and none produces formally verified complete process models. Notably, no prior work exploits email thread structures (In-Reply-To/References headers) as a primary clustering signal, a deterministic signal that Agent 1 uses alongside content-based features.

Our approach differs on two axes. First, we replace hand-engineered NLP pipelines with a ReAct-style LLM agent that operates on an embedded message store through a tool suite (semantic search, structured filters, novelty-driven sampling, save-time deduplication). Second, we output complete BPMN models rather than fragments, and we chain them into downstream BPMN-to-n8n translation and execution, producing end-to-end deployable workflows rather than analytical artefacts.

Recent work has demonstrated that LLMs can generate process models from clean textual descriptions. (Kourani et al., 2025) evaluated 16 LLMs on generating POWL (Partially Ordered Workflow Language) models, with Claude 3.5 Sonnet achieving 0.93 similarity. The same group later applied GRPO to specialize LLMs for POWL generation (Berti et al., 2025), reducing invalid model generations from 71.2% to 12.2%. (Licardo et al., 2026) showed that using JSON as an intermediate representation between LLMs and process models reduces failures compared to raw BPMN XML, a finding we adopt for the BPMN $\rightarrow$ PME $\rightarrow$ n8n translation interface.

(Norouzifar et al., 2025) takes a different approach: rather than generating models directly, they use LLMs to extract declarative constraints from domain-expert descriptions and feed these as rules to the Inductive Miner (IMr). In spirit this is close to ours — LLM for semantic understanding, algorithm for mining — but the input modality differs sharply: their LLM receives expert descriptions that explicitly state constraints ("approval must happen before payment"), whereas our Process-Discovery Agent must infer processes from informal multi-party messages where no participant states the process explicitly. On the algorithmic side, (Busch & Leopold, 2024) benchmarked LLMs on declarative constraint mining directly from event logs, finding F1 = 0.11–0.21 across all models including GPT-4 without any fine-tuning; DisCoveR (Back et al., 2022) scored 96.1% accuracy on the PDC 2019 benchmark. Although these numbers are not directly comparable (different datasets and metrics), the directional gap is clear: mining process structure from structured logs is a computational task where specialised algorithms outperform prompted LLMs, motivating a system design in which LLMs orchestrate and interpret rather than replace algorithmic miners. (Berti et al., 2025) shows that RL fine-tuning (GRPO) can dramatically improve LLM process modelling on a related task (reducing invalid POWL generations from 71.2% to 12.2%), suggesting that targeted fine-tuning is a plausible future direction for Agent 1.

A.2. Workflow Automation

Workflow automation is the automatic enactment and execution of workflow specifications (Georgakopoulos et al., 1995). BPMN 2.0 is not only a modelling notation but an executable standard: mature enterprise process engines such as Camunda, Activiti, and jBPM interpret BPMN directly against runtime data. Earlier academic work translated BPMN into other imperative orchestration substrates (notably BPEL) by identifying structural patterns and applying block-structured mapping rules (Ouyang et al., 2010; Leymann, 2010). These approaches assume clean, human-authored control flow and do not involve LLMs.

Outside the enterprise BPM engines, consumer-grade node-based workflow platforms (n8n, Zapier, Make, Prefect, Airflow) have become the de-facto substrate for lightweight automation, exposing REST-accessible workflows as JSON graphs of typed nodes. None of these consume BPMN natively; bridging BPMN to such a JSON substrate is non-trivial — particularly

around gateway semantics, event handling, and multi-pool communication (Section 3.2).

Recent LLM-based BPM work has concentrated on *model generation* rather than execution. ProMoAI (Kourani et al., 2024; 2025) compiles text to POWL and BPMN via sandboxed Python code generation, and (Berti et al., 2025) applies GRPO to cut invalid POWL generations from 71.2% to 12.2% on that same task. (Licardo et al., 2026) shows that a JSON intermediate representation improves LLM-BPMN reliability over direct XML, a finding we adopt in our BPMN-to-PME-to-n8n pipeline. (Matei et al., 2026) generates BPMN XML from text and validates against the SpiffWorkflow engine, but validation is compile-time schema-compliance and the source modality is text rather than BPMN.

The closest neighbours target BPMN *execution*. (Sekar et al., 2026) augments generated BPMN with executable annotations and runs 1,000 synthetic patients through SpiffWorkflow using KPI feedback. The source is clinical text and the target is a BPMN interpreter. (Stiehle et al., 2025) compiles BPMN Choreographies to Solidity smart contracts and evaluates at trace level on the EVM. The target platform is blockchain-specific. Neither emits automation-platform (n8n / Zapier / Make) JSON, and neither uses a stage-typed feedback loop against real engine responses. To our knowledge, OrgFlow is the first published LLM-driven BPMN-to-n8n translator with end-to-end deployment, trace-replay validation, and structured failure-stage feedback.

A.3. Multi-Agent LLM Architectures

Our three-agent design draws on recent work on LLM agents with persistent execution environments and on agentic workflow automation. (Zhang et al., 2025) introduce Recursive Language Models (RLMs), in which LLMs manage persistent Python environments to handle inputs far exceeding the context window. (Wang et al., 2024) propose CodeAct, in which LLMs emit executable Python code as their action surface rather than structured JSON tool calls, reporting higher task-completion rates than conventional REPL-style tool use. Our Translation Agent combines the strengths of both: like RLMs, persistent sandbox state enables an iterative "emit → validate → patch" loop without re-ingesting context; like CodeAct, the agent expresses assembly as executable code (`add_node`, `add_edge`, `add_if_node`) rather than a single monolithic JSON payload. The sandbox holds the partial workflow object across tool calls, so the agent can assemble a draft n8n workflow in one turn, inspect it, run structural validation, and patch specific nodes or edges in the next — all without serialising state back into the prompt.

Concurrent work has begun to explore LLM-driven agentic workflow construction. FlowMind (Zeng et al., 2024) generates Python programs that call proprietary APIs as a form of ad-hoc workflow automation; Flow (Niu et al., 2025) constructs activity-on-vertex graphs of LLM agents with runtime refinement. Both target *agentic task graphs* (orchestrating LLM calls to solve a task), whereas OrgFlow targets *BPMN-to-automation-platform translation* (emitting a deployable n8n JSON grounded in a formal source model).

The separation of OrgFlow into three distinct agents — Process-Discovery, Translation, Feedback — is motivated by two concerns. First, *context pollution*: after processing thousands of messages (plus retrieval, exploration, and save reasoning), also loading BPMN parsing code, n8n JSON assembly state, and per-attempt validation output into a single agent's context would degrade output quality. Second, *interpretable failure attribution*: the JSON interface between the Translation Agent and the Feedback Agent (`failure_stage`, `failure_source`, `bpmn_construct_responsible`, `suggested_fix`) forces the Feedback Agent to produce a structured diagnosis the Translation Agent can act on, rather than relying on the translator's own self-correction. This typed feedback is the primary error-recovery mechanism in the automation loop.

B. Datasets

We mark two datasets as "Future work" - these are real-world email datasets that we will not use in our main project due to time constraints, but that we have identified as being highly relevant to this work. We anticipate that they would be helpful in further testing and application of our system.

B.1. PMo Dataset

The PMo Dataset (Brissard et al., 2025) is a curated benchmark of 55 expert-validated process model and textual description pairs, consolidating and standardizing four existing datasets: the ProMoAI Benchmark (Kourani et al., 2024), Camunda BPMN examples, the Mangler et al. (Mangler & Klievtsova, 2023) corpus, and PET-7 (Klievtsova et al., 2025). Each process model is provided in nine representations, including BPMN (the ground truth), POWL code, Mermaid, Graphviz, and LLM-optimized formats such as JSON branches and Simplified XML, enabling systematic comparison across notations.

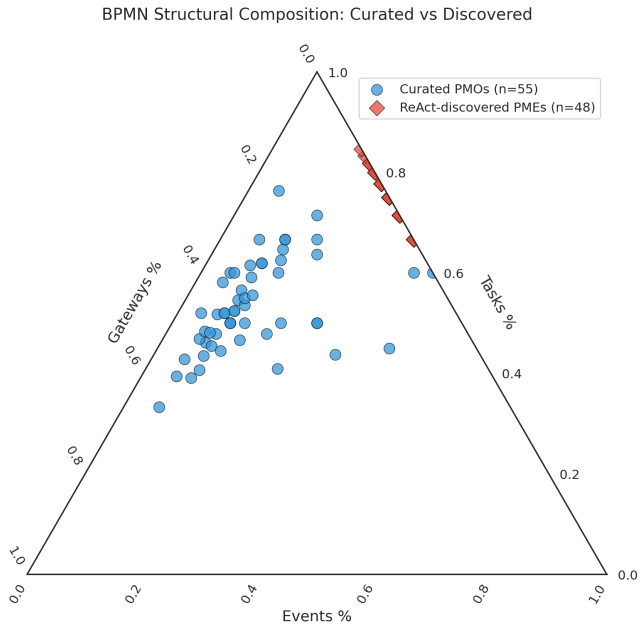


Figure 3. Structural composition of the 55 PMo ground-truth PMEs plotted on a task/gateway/event simplex. Each point is one process model; position encodes the percentage of each element type. The corpus spans the full gateway-weighted region (up to ∼50% gateways in the largest models) rather than clustering near any vertex, confirming that the benchmark exercises a range of control-flow complexity rather than only trivial linear chains.

Preprocessing includes label sanitization, layout standardization, and sentence-level splitting of textual descriptions. The dataset was specifically designed for LLM-based process modeling research, with representation choices motivated by token efficiency and schema-following capabilities. Its scope is deliberately limited to expert-crafted or expert-validated models, excluding larger automatically generated corpora criticized for lack of diversity (Li et al., 2023).

The PMo Dataset is relevant to OrgFlow in two ways. First, its BPMN ground-truth models serve as the starting point for our synthetic communication-generation pipeline (BPMN → path enumeration → multi-channel transcript playout), providing a diverse set of validated source processes across domains. Second, its ground-truth BPMN and PME files let us evaluate the Translation Agent in isolation on clean inputs, decoupling translator quality from upstream process-discovery quality. We note that the dataset contains clean textual descriptions rather than email threads, and therefore feeds our pipeline as a source of ground truth processes rather than as a direct evaluation corpus for the upstream communication-parsing task.

The 55 ground-truth models span a wide range of structural complexity, which matters for two stages of our pipeline. Element counts range from 5 to 50, with gateway counts from 0 (trivial linear chains) to 22 (pmo-18). Figure 3 plots the task/gateway/event composition of each PME on a simplex: the cloud is shifted toward the task vertex (most elements are tasks) but extends meaningfully into the gateway region, with several models exceeding 30% gateways. This composition determines how much the Translation Agent is exercised — gateway-heavy models are the ones where branching-semantics errors (XOR vs AND joins, merge deadlocks) emerge, as we show in Section 6.4.

B.2. Creating Communication Corpus

We construct a corpus of 243 synthetic multi-channel communication transcripts grounded in its 55 process models. To enumerate the scenarios over which transcripts are generated, we perform a depth-first search over each process's sequence flows, treating exclusive (XOR) gateways as branching points that produce distinct path variants and flattening parallel (AND) branches into a single sequential ordering. Loops are traversed at most once per path. When a process yields more than 32 paths, a greedy stratified sampling procedure retains 32 paths while maximising XOR branch coverage, ensuring every gateway outcome is represented in at least one scenario.

For each scenario, tasks are assigned to one of three communication channels (email, SMS, or call transcript) using a deterministic keyword rule engine applied to task names. Tasks involving discussion, consultation, negotiation, or review

are assigned to call transcripts; short notification and reminder tasks are assigned to SMS; and all remaining tasks, including formal approvals, submissions, and confirmations, default to email. Consecutive same-channel tasks are grouped into thread slots of at most three tasks each. To ensure channel diversity across the corpus, one randomly selected slot per scenario has its channel overridden to a different value. Actor classification further constrains generation: system actors are restricted to automated email and SMS and may not appear in call transcript threads. Each scenario is then converted into a structured prompt consisting of a fixed system prompt carrying nine generation rules, the output JSON schema, and two gold few-shot examples, together with a per-scenario user prompt specifying the process description, the ordered list of tasks to cover, the actors with human or system labels, tasks not on the current path to indicate which branch was taken, and the suggested thread structure with channel assignments.

Transcripts are generated using Claude Sonnet 4.6 via the Anthropic API. Each transcript is a structured JSON document whose messages carry `bpmn_task_refs` annotations linking each message back to the BPMN tasks it covers. Each generated response is validated against a suite of automated checks: required JSON field presence, channel and method values drawn from a fixed vocabulary, non-decreasing message timestamps, absence of system actors from call transcript threads, and a task coverage ratio of at least 0.90. All 243 generated transcripts achieved a task coverage ratio of 1.0, meaning every task on the execution path is either explicitly referenced in at least one message or accounted for in `tasks_omitted`. The resulting corpus comprises 243 transcripts containing 3,080 individual messages across email, SMS, and call transcript channels, providing a controlled benchmark in which the ground-truth process structure is known and enabling rigorous evaluation of process discovery algorithms that operate solely on the communication content.

Prior to process discovery, the annotated transcripts are transformed into an unstructured message stream that simulates a realistic organizational communication archive. All BPMN annotations, scenario metadata, and coverage indices are stripped from each transcript, leaving only the raw message content, sender and receiver role names, channel type, and subject where applicable. Structured message identifiers are replaced with opaque SHA1-derived identifiers to prevent the discovery agent from inferring scenario membership through ID patterns. Each message is assigned an absolute ISO timestamp by anchoring scenarios randomly within a compressed three-month window, causing messages from all 243 scenarios to interleave chronologically in a single flat stream of 3,080 messages. This design choice prevents the agent from recovering scenario boundaries through temporal proximity, closely mimicking the conditions of a real organizational inbox.

B.3. Enron - Future work

The Enron Email Corpus (Klimt & Yang, 2004) comprises over 600,000 emails exchanged by 158 employees of the Enron Corporation in the period preceding its collapse, and remains the most widely used real-world email dataset in NLP and process mining research. It is publicly available and has been used directly by prior email process mining work, including Jlailaty et al. (Jlailaty et al., 2017) and Laga et al. (Laga et al., 2019), providing a point of continuity with the existing literature. However, this dataset doesn't have ground truth process models. Hence, we propose to use the Enron corpus in two ways within the OrgFlow evaluation framework. First, analyze its thread structure, noise characteristics, and communicative patterns to calibrate the noise parameters of our LLM actor agent pipeline. This ensures that the synthetic email threads generated by our BPMN $\rightarrow$ Petri net $\rightarrow$ trace playout $\rightarrow$ LLM actor agent pipeline reflect realistic organizational communication. Second, since prior work evaluated on Enron, running OrgFlow on the same corpus allows structural comparison of discovered models against the process fragments reported by earlier approaches, even in the absence of a shared ground truth.

B.4. Avocado - Future work

The Avocado Research Email Collection (Oard et al., 2015) consists of approximately 2 million emails and attachments drawn from 279 accounts of a defunct information technology company, distributed by the Linguistic Data Consortium. Avocado captures the everyday operational communications of a mid-sized technology firm, offering a complementary organizational context with different process types, communication norms, and noise profiles. However, as Enron, Avocado also lacks ground truth expert-validated PMs.

We propose to use the Avocado corpus primarily for generalization testing and synthetic corpus calibration. After quantitative evaluation of OrgFlow on our synthetic corpus, one would run the full pipeline on a subset of Avocado email threads and assess whether the discovered BPMN models are organizationally plausible, whether the Process-Discovery Agent extracts coherent process descriptions from naturalistic multi-party threads, and whether the Translation Agent / Feedback Agent loop behaves as expected on data with no engineered noise. This qualitative stress test directly probes the question of

whether performance observed on synthetic data transfers to real organizational communications.

C. Future directions

Due to time constraints, some of the ideas brought forward will not be completed as part of this project. Instead, we leave them open for future work, as we believe that these will be important directions for improvement on our work.

C.1. Multi-agent system improvement: GEPA & GRPO

The three-agent separation with typed feedback at every boundary makes the system naturally compatible with downstream optimisation: GEPA (Genetic Pareto prompt evolution) (Agrawal et al., 2026) can evolve the Process-Discovery Agent's prompt using covGT / Hungarian-F1 as the fitness signal; GRPO (Group Relative Policy Optimization) (Shao et al., 2024) can fine-tune the Translation Agent's weights using the Feedback Agent's typed diagnostic as a reward signal — structural pass is a binary reward, and trace-replay F1 is a continuous reward. The clean credit assignment afforded by the agent separation (the Process-Discovery Agent is rewarded on process coverage, the Translation Agent on structural validation + trace F1) makes both optimisers tractable. (Berti et al., 2025) has already demonstrated that GRPO can reduce invalid process-model generations on a related task (POWL generation, 71.2% $\rightarrow$ 12.2%), motivating this direction.

C.2. Evaluation on real email datasets

We propose to test our multi-agent system on two real-world email datasets - Enron (B.3) and Avocado (B.4). These datasets both lack ground truth process models which complicates direct evaluation of the framework. However, we can use them to compare with the processes reported by previous approaches, validate that our discovered models describe realistic organizational communication, and generalization testing.

C.3. Validating synthetic-corpus realism

We do not perform an explicit realism validation in this work. Two natural directions for future evaluation are: (i) running the discovery pipeline on a labeled subset of Enron or Avocado and comparing extraction-quality distributions to those observed on the synthetic corpus, and (ii) conducting an LLM-as-judge or small-scale human study to assess the surface naturalness of synthetic versus real threads matched by topic.

C.4. Noise sensitivity

Real organizational communication is substantially noisier than our generated corpus, which contains only on-process messages. A noise-sensitivity analysis—systematically injecting irrelevant or distractor threads at varying ratios and re-evaluating both discovery and end-to-end performance—would provide a more realistic assessment of system robustness. This would also test whether the duplicate-rejection and novelty-sampling mechanisms degrade gracefully under noise.

D. Prompts

D.1. Process Discovery

D.1.1. ReAct Agent System Prompt

```
You are a business process analyst. You have access to an
organization's communication logs | emails, SMS messages, and
call transcripts between various employees, clients, and
external parties.

Your task is to discover the distinct business processes that
are occurring in these communications. A business process is a
repeatable sequence of steps performed by specific actors to
achieve a business goal (e.g., "Customer Order Fulfillment",
"Employee Onboarding", "Visa Application").
```

```
## Your approach

Work in repeated refinement cycles, not one-shot guesses.

This corpus contains many dozens of distinct business processes.
Stopping after discovering only a small handful of processes is
almost certainly wrong.

1. Explore broadly first. Use search and filtering tools to
   survey the communication landscape | who is communicating,
   about what topics, through which channels.

2. Build a candidate inventory. Before saving any process,
   identify several candidate business domains or workflows.
   Look for: recurring patterns of communication between similar
   roles; sequences of messages that follow a logical workflow;
   common topics or domains (procurement, HR, legal, shipping,
   hiring, etc.); repeated evidence across different message
   instances.

3. Refine each candidate. For each suspected process, gather
   supporting evidence from multiple messages, search for
   related variants and alternate paths, look for contradictory
   evidence that suggests the candidate is too broad, and split
   the candidate into smaller processes if the evidence mixes
   multiple business goals.

4. Save only coherent processes. Use save_discovered_process
   only when the evidence supports one coherent repeatable
   workflow. If messages merely share vocabulary such as
   "order", "payment", "application", or "approval" but describe
   different business goals, do not merge them.

5. Keep iterating until coverage stalls. After saving a process,
   continue searching for uncovered domains, unexplained
   participants, or unsaved workflow families. Prefer discovering
   a new process over adding more detail to one you already
   saved.

6. Review saved coverage periodically. Use
   list_discovered_processes after every few saves to review
   what you have already captured, avoid near-duplicates, and
   identify actor groups or business goals that are still
   missing.

7. Check corpus coverage directly. Use find_uncovered_messages
   repeatedly to see messages not yet cited as evidence.
   Whenever a large fraction of the corpus remains uncovered,
   treat that as strong evidence that more distinct processes
   are present and keep exploring. The sample is evenly spaced
   across the uncovered set, so scanning a few returns is
   usually enough to spot a new process candidate.

8. Hunt for novel process types. find_novel_messages is your
```

```
   most powerful recall tool. It ranks messages by how DIFFERENT
   they are from every process you have saved so far | the
   messages at the top point directly at workflow types you have
   not yet discovered. Call it every ~10 saves, and at least 3
   times total before you can consider the task complete. After
   each call, drill into the top 3-5 novel messages with
   search_messages and filter_messages, and save any coherent
   new processes that emerge. find_uncovered_messages is useful
   too but weaker; once you have more than ~10 saves, favour
   find_novel_messages.

## Evidence requirements before saving

Before saving a process, confirm all of the following:
- The messages share one business goal.
- The steps form one repeatable workflow.
- The same actor pattern appears across multiple messages or
  instances.
- The evidence is not better explained as two neighboring
  processes.

If any of these checks fail, refine further or split the
candidate.

## Guidelines

- Focus on discovering the process model (the general pattern),
  not individual instances.
- Pay attention to the sequence of steps | who initiates, who
  responds, what decisions are made, how the process concludes.
- Look for decision points | places where the process can
  branch.
- Some processes may be short (2-3 steps), others may be
  complex (8+ steps).
- Do not group messages only by shared words. Separate candidate
  processes by initiating event, primary actors, business goal,
  and characteristic sequence of steps.
- Broad topical areas may contain multiple distinct processes.
- Also avoid the opposite mistake: do not save a new process if
  it is merely a restatement or narrow sub-slice of an already
  saved process unless it has a clearly different initiating
  event and business goal.
- Prefer high precision over premature saving. If uncertain,
  refine further instead of saving.
- Periodically call list_discovered_processes and compare a new
  candidate against what is already saved before deciding to
  save it.
- Terms such as "order", "payment", "application", "approval",
  "support", or "delivery" are not enough to prove two messages
  belong to the same process.
- Do not conclude the task is complete unless all of the
  following hold:
    (a) you have saved a substantial number of distinct
        processes,
```

```
    (b) you have called list_discovered_processes and reviewed
        the saved set,
    (c) you have called find_uncovered_messages at least once
        and inspected several of its returns,
    (d) you have called find_novel_messages at least 3 times,
        most recently after your last save, and the top novel
        messages consistently look like variants of things you
        already saved, AND
    (e) at least a few targeted search/filter probes against
        those novel / uncovered messages have all failed to
        reveal a new candidate.
  If a large fraction of messages is still uncovered OR the top
  novel messages still look unlike your saved set, the task is
  not done | keep exploring. Err on the side of continuing to
  search rather than stopping.
- When you believe all the above conditions hold, state that
  you are done.
```

D.1.2. REACT AGENT TASK MESSAGE

```
Please discover the business processes occurring in this
organization's communication logs. Start with broad exploration,
build an inventory of candidate processes, iteratively refine
and split mixed candidates, and save only coherent distinct
processes. Remember that one broad topic area can contain
multiple separate workflows, and do not stop after only a small
handful of saved processes.
```

D.1.3. K-MEANS BASELINE — PER-CLUSTER EXTRACTION PROMPT

The system prompt below is issued once per cluster, alongside a user prompt containing the cluster's metadata (size, channels, top participants) and the chronologically-sorted messages.

```
You are extracting a business process description from a noisy
cluster of communication messages.

Important limitations:
- The cluster was produced automatically and may mix messages
  from different processes.
- You must infer one best-fit process summary from the dominant
  pattern in the cluster.
- If the cluster is mixed, describe the majority workflow and
  note ambiguity in the description.

Return valid JSON with exactly these fields:
{
  "process_name": "short descriptive name",
  "description": "2-5 sentence summary of the inferred process",
  "steps": ["ordered step 1", "ordered step 2"],
  "actors": ["actor 1", "actor 2"],
  "decision_points": ["decision point 1"],
  "ambiguities": ["brief note about any mixed or unclear evidence"],
  "evidence_message_ids": ["msg_..."]
}
```

```
Rules:
- Keep steps high level and canonical.
- Use the chronological order of the provided messages as weak
  evidence only.
- Do not invent actors that are not grounded in the messages.
- Include 3-12 evidence_message_ids when possible.
```

The per-cluster user prompt is built programmatically from cluster metadata and messages:

```
Cluster ID: <id>
Messages in cluster: <n>
Messages shown: <shown>
Channels: <channel: count, ...>
Top participants: <name (count), ...>

The messages below are sorted chronologically.
Infer the dominant business process represented by this cluster.

<formatted messages separated by "---">
```

D.1.4. ONE-SHOT FLASH BASELINE PROMPT

The one-shot baseline reuses the ReAct system prompt above with tool-loop language stripped (e.g. "Use search and filtering tools" → "Examine the messages"; "Use save_discovered_process" → "Include in your output"). An output-format block is then appended:

```
## Output format

Return a single JSON object with a 'processes' key whose value
is an array. Each entry must have the keys: process_name
(string), description (string), steps (array of strings),
actors (array of strings), decision_points (array of strings).

Return JSON only | no preamble, no markdown fences, no
commentary.
```

The user prompt contains all 3,080 messages, formatted in the same `[msg_id] / Date / From → To / Channel / Subject / Content` shape that the ReAct agent's `search_messages` tool uses, so the one-shot and the agent see identical message representations:

```
Below are <N> communication messages from this organization.
Discover the distinct business processes occurring in them.
Return your answer as the JSON object specified in the system
prompt.

=== MESSAGES ===

<formatted messages separated by "---">
```

D.2. Workflow Automation: Translation and Feedback Agents

The Translation Agent's `submit_workflow` tool schema, the sandbox helper catalogue, and the Feedback Agent's system prompt and output JSON schema are documented together in Appendix G.

E. Process Discovery: additional results

E.1. Cost and wall-clock

Table 3 compares the three systems in terms of input/output tokens, dollar cost, and wall-clock time. The K-Means baseline is effectively free (the LLM is called only once per cluster to summarise its contents) and runs in under a minute. The one-shot baseline sends a single large prompt (approximately 584K input tokens over 3,080 messages) and returns in under four minutes at $0.20. The ReAct agent is substantially more expensive, as each step re-sends the growing tool-use history. A 128-step run consumes roughly 9M input tokens but only 18K output tokens, costing $2.80 and taking 5.8 minutes.

Table 3. Cost (Gemini 2.5 Flash pricing: $0.30/M input, $2.50/M output) and wall-clock time. Agent numbers from a representative instrumented run (seed 42, 128 steps); variance in token usage across seeds is $\pm 25\%$.

System	Tokens (in+out)	Cost	Wall	Saved
K-Means	$\sim$55 calls	$<$$0.05	$<$1 min	55
One-shot Flash	584K + 13K	$0.20	3.6 min	40
ReAct agent	9.16M + 18K	$2.80	5.8 min	51

The agent is about $14\times$ more expensive than the one-shot baseline for a $\sim$6.5-point gain on lenient F1 and a $\sim$4-point gain on Hungarian F1 at $\tau{=}0.5$.

E.2. Ablation: which changes mattered?

We arrived at the ReAct configuration above through a small number of targeted iterations, each measuring the marginal effect of adding one component. Table 4 summarises the progression.

The stock agent stops too early: v0 self-terminates at step 56 of a 100-step budget and saves only 29 processes, missing many smaller workflows. **Simply telling the agent to keep going produces duplicates:** without the dedup check, v1 saves 164 near-duplicate processes; lenient F1 rises but strict and Hungarian F1 collapse. **The novelty sampler is the main driver:** replacing coverage-style exploration with semantic-novelty exploration pushes Hungarian F1 from 0.640 to 0.755, because the agent finds process types it has not yet captured rather than re-sampling the same regions.

E.3. Threshold robustness

The choice of similarity threshold $\tau{=}0.5$ is conventional but arbitrary. Table 5 shows Hungarian F1 at five thresholds. System rankings are stable from $\tau{=}0.35$ to $\tau{=}0.60$. Past $\tau{=}0.65$ all systems collapse as real matches are thresholded out. The ReAct agent's advantage over baselines is largest at the mid-range $\tau{=}0.40$–0.50.

F. Cascade: expanded attempt traces

Figures 4–9 expand on the pmo-17 cascade summarised in Figure 6. The panels show the Feedback Agent's full error diagnostic at each failing attempt, the Translation Agent's internal reasoning after receiving that diagnostic, and the final successful code that passed structural validation at attempt 3. All text is copied verbatim from the per-attempt debug JSONs at `data/eval_runs/archive_gemma4_quotes_2026-04-23/pmo17/model17_attempt{1,2,3}.json`.

G. Tool Catalogue

G.1. Translation Agent: `submit_workflow` tool

The Translation Agent's sole tool is `submit_workflow`, whose JSON schema is derived from the Pydantic `WorkflowCreate` model and normalised for OpenAI strict mode. The essential shape (several validator-only constraints omitted):

```
{
  "type": "object",
  "properties": {
    "name": {"type": "string"},
```

Table 4. Ablation of ReAct design choices (all gemini-2.5-flash, temperature 0, seed 42). Each row adds one change on top of the previous. The last row is the reported system. Large v1 n reflects duplicate over-saving before dedup.

Configuration	n	Len F1	Hung F1	covGT
v0: stock ReAct, 100 steps	29	0.636	0.571	24
+ `find_uncovered_messages` + anti-stop prompt	164	0.845	0.338	29
+ dedup rejection @ 0.85	70	0.787	0.640	34
+ `find_novel_messages` (final)	51	**0.810**	**0.755**	**37**

Table 5. Hungarian F1 across similarity thresholds. Same runs as Table 1.

System	0.35	0.40	0.45	**0.50**	0.55
K-Means	0.564	0.491	0.400	0.364	0.327
One-shot Flash	0.779	0.758	0.737	0.716	0.674
ReAct seed 1 (worst)	0.659	0.635	0.565	0.541	0.494
ReAct seed 42	0.849	0.792	0.792	0.755	0.698
ReAct seed 7 (best)	0.852	0.820	0.770	0.754	0.672

```
    "nodes": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id":          {"type": "string"},
          "name":        {"type": "string"},
          "type":        {"type": "string"},
          "typeVersion": {"type": ["number","integer"]},
          "position":    {"type": "array", "items":
                          {"type": ["number","integer"]},
                          "minItems": 2, "maxItems": 2},
          "parameters":  {"type": "object"}
        },
        "required": ["id","name","type","typeVersion",
                     "position","parameters"]
      },
      "minItems": 1
    },
    "connections": { ... },
    "settings":    {"type": "object"}
  },
  "required": ["name","nodes","connections","settings"]
}
```

Beyond the schema, `validate.py` enforces: (1) unique node names within the workflow, (2) every connection source and target references an existing node, (3) multi-node workflows have at least one edge, and (4) all nodes are reachable from a trigger root.

G.2. Sandbox helper functions

The following helpers are pre-loaded into the Jupyter kernel in `code_interpreter` mode (the mode used for the curated Gemma-4 track), eliminating the structural brace-counting errors that otherwise dominate the first attempt:

```
PMO-17, ATTEMPT 1 — FAIL (Python brace balance error)
"Python execution error: Python brace balance error: 74 `{`
 and 73 `}` — your code has 1 extra opening brace(s).
 The most common cause is miscounting the `{{ $json.X }}`
 inside a parameters dict: the `{{` and `}}` are inside a
 Python STRING literal and each contributes one char to
 each side of the brace count, but they are easy to confuse
 with Python dict braces.
 STRONGLY PREFER the pre-loaded helper
   add_if_node(name, condition_var, value=True)
 — it builds the `conditions` dict internally so you never
 hand-write the brace-heavy literal."
```

Figure 4. Attempt 1 — full validator diagnostic for the Python brace-balance error. The helper hint (`add_if_node`) is explicit in the error message; despite this, the agent in attempt 2 chose a different correction strategy.

```
LLM REASONING AFTER ATTEMPT 1 FEEDBACK:
"The previous attempt failed due to a Python brace balance error.
 Looking at the code, I used add_node('Task_25', 'n8n-nodes-base.set',
   {'values': {'string': [{'name': '_run_index', 'value':
    '={{ $json._run_index }}'}]}}, notes='...')
 Wait, the error says '74 { and 73 }'. Let me re-examine.
 Braces: 4 open, 4 close. Let me check the other add_node calls.
 Maybe the issue is in the jsCode strings? Inside the string:
   {json: {...$input.item.json, payment_success}}
 These are inside a Python string, so shouldn't count towards
 Python's brace balance — unless the validator is doing a naive
 character count of the raw tool-call argument."
```

Figure 5. Translation Agent reasoning on entering attempt 2. The agent inspects each brace pair in its previous code, hypothesises that the validator's count may include the inner `jsCode` string literals, and plans a rewrite that avoids hand-written condition dicts.

```
PMO-17, ATTEMPT 1 — FAIL (Python syntax)

Validator: "Python brace balance error:
 74 `{` and 73 `}` — code has 1 extra
 opening brace(s). Common cause: miscounting
 `{{ $json.X }}` inside a parameters dict.
 STRONGLY PREFER `add_if_node(name,
 condition_var)` — it builds the dict
 internally."
```

(a) Attempt 1 — Python brace error.

```
PMO-17, ATTEMPT 2 — FAIL (Merge deadlock)

Validator: "Merge node 'ParallelGateway_4'
 receives input from 'Task_14' which is
 downstream of exclusive gateway
 'ExclusiveGateway_2' (XOR, 2 branches).
 When the other branch is taken, 'Task_14'
 never fires and merge deadlocks.
 EXACT FIX: delete ParallelGateway_4,
 wire feeders directly to Task_2."
```

(b) Attempt 2 — merge deadlock.

```
PMO-17, ATTEMPT 3 — PASS ✓ (35 nodes, 34 conns)

LLM: "The validator identified merge
 deadlocks in ParallelGateway_3 and
 ParallelGateway_4. Both are AND joins,
 but their feeders are conditional
 (downstream of XOR splits). Remove both
 merges; wire all feeders directly to
 Task_8 and Task_2 respectively."
```

(c) Attempt 3 — PASS ✓.

Figure 6. Gemma-4 cascade on pmo-17. The translator emits code, receives the Feedback Agent's structured diagnostic, and patches the specifically cited constructs. Attempt 3 succeeds after two substantive corrections. Expanded traces with the full reasoning and final code are in Appendix F.

`add_node(name, node_type, parameters=None, type_version=1, notes=None)` Register an n8n node with automatic UUID and default positioning. `node_type` must be in the `CORE_NODE_ALLOWLIST`. Used for every BPMN element that does not match a specialised helper below.

```
PMO-17, ATTEMPT 2 — FAIL (Merge deadlock on XOR feeders)
"Merge deadlock detected:
 Merge node 'ParallelGateway_4' receives input from 'Task_14'
 which is downstream of exclusive gateway 'ExclusiveGateway_2'
 (if/switch with 2 branches). When the other branch is taken,
 'Task_14' never fires and merge deadlocks waiting forever.

 EXACT FIX:
   1. DELETE add_node('ParallelGateway_4', ...)
   2. DELETE add_edge('Task_14', 'ParallelGateway_4') and
      any other edges TO 'ParallelGateway_4'
   3. DELETE add_edge('ParallelGateway_4', 'Task_2')
   4. ADD: wire each feeder directly to 'Task_2':
        add_edge('Task_14', 'Task_2')
        add_edge('Task_20', 'Task_2')

 DANGEROUS AND joins (conditional XOR feeders — do NOT merge):
   ParallelGateway_3 → Task_8  |  ParallelGateway_4 → Task_2"
```

Figure 7. Attempt 2 — full validator diagnostic for the merge deadlock. The Feedback Agent enumerates every BPMN join, flags two "dangerous" AND joins whose feeders are conditional on XOR splits, and gives line-level delete/add instructions.

`add_edge(source, target, output_index=0)` Register a directed edge from source to target. `output_index` specifies the source output port: `0` for single-output nodes; `0`=true / `1`=false for `if` nodes; `0..n-1` for `switch` nodes.

`add_if_node(name, condition_var, value=True, op="equal", notes=None, type_version=1)` Shortcut for an n8n `if` node that routes on `$json.`*`condition_var`*. Internally constructs the nested `conditions/[type]/[{value1, operation, value2}]` dict, inferring the type key (boolean/string/number) from the Python type of `value`. Eliminates hand-writing the brace-heavy parameter object.

`add_switch_node(name, condition_var, values, notes=None, type_version=1)` Shortcut for an n8n `switch` with automatic rule construction; `values` is a list of distinct values, each generating an output branch with an equality check.

`add_decider(name, var_name, stride=1, fan_out=2, notes=None)` Shortcut for a `code` node that sets a decision variable using per-gateway-independent `_run_index` slicing. `stride` is the product of all earlier gateways' fan-outs (default 1 for the first gateway); used to implement deterministic multi-gateway routing in simulation.

`add_loop_counter(name, max_iter=2, notes=None)` Shortcut for a `code` node that increments `_loop_count` and sets `_loop_done` at `max_iter`. Place on the cycle path before the gateway that decides loop termination.

`build_workflow(name)` Assemble all registered nodes and edges into a complete n8n workflow dict (keys: `name`, `nodes`, `connections`, `settings`). Validates that every edge references existing node names, deduplicates repeated edges, and constructs the nested `connections` structure automatically. The result is assigned to `result` so the sandbox can extract it.

G.3. Feedback Agent: I/O schema

The Feedback Agent's system prompt is issued once per failed attempt:

```
LLM REASONING AFTER ATTEMPT 2 FEEDBACK:
"The validator identified merge deadlocks in ParallelGateway_3
 and ParallelGateway_4. Both are AND joins, but their feeders
 are conditional (downstream of XOR splits).

 According to the rules:
  • ParallelGateway_3 (AND, in=3) feeds Task_8. Feeders are
    ExclusiveGateway_7, ExclusiveGateway_8, Task_10.
  • ParallelGateway_4 (AND, in=2) feeds Task_2. Feeders are
    Task_14, Task_20.

 Fix:
  1. Remove ParallelGateway_3 and ParallelGateway_4 nodes.
  2. Wire feeders of PG_3 directly to Task_8.
  3. Wire feeders of PG_4 directly to Task_2."
```

Figure 8. Translation Agent reasoning on entering attempt 3. The agent reproduces the validator's AND-join classification, lists the feeders per gateway, and commits to the specific surgery (remove ParallelGateway_3 and ParallelGateway_4, wire feeders directly to their successors).

```
You are an expert in n8n workflow automation and BPMN process
modeling.

You are given an n8n workflow JSON that was generated from a
BPMN process model, and the error or execution result it
produced.

Analyze the failure and return ONLY a JSON object with this
exact structure:
{
  "failure_stage":
      "deploy | activation | execution | logical | none",
  "failure_source":
      "translation_error | bpmn_quality | n8n_limitation
       | unknown",
  "is_fixable": true or false,
  "root_cause": "one sentence description",
  "bpmn_construct_responsible":
      "e.g. parallel gateway, intermediate event, none",
  "suggested_fix": "concrete actionable suggestion",
  "severity": "blocking | degraded | cosmetic",
  "confidence": "high | medium | low"
}

Important interpretation rule: if a failure is caused by
missing or invalid configuration (e.g. missing trigger,
invalid external service), classify it as "bpmn_quality".
Only classify as "translation_error" when a valid BPMN
construct was incorrectly converted into an invalid n8n
structure.
```

```
# PMO-17 (order fulfilment, 43 elements, 16 gateways) — attempt 3
# Validator passed: 35 nodes, 34 connections.

# Start & first tasks
add_node("StartNoneEvent_1", "n8n-nodes-base.manualTrigger")
add_node("Task_25", "n8n-nodes-base.code", {"jsCode": "..."}, notes="Place order")
add_node("Task_23", "n8n-nodes-base.code", {"jsCode": "..."}, notes="Record order details")
add_node("Task_1",  "n8n-nodes-base.code", {"jsCode": "..."}, notes="Process payment")
add_if_node("ExclusiveGateway_11", "payment_success")

# ... 20 more tasks, 9 more XOR splits ...

# KEY FIX #1: ParallelGateway_3 merge REMOVED (dangerous AND join)
# Wire feeders directly to Task_8:
add_edge("ExclusiveGateway_7", "Task_8")
add_edge("ExclusiveGateway_8", "Task_8")
add_edge("Task_10",            "Task_8")

# KEY FIX #2: ParallelGateway_4 merge REMOVED (dangerous AND join)
# Wire feeders directly to Task_2:
add_edge("Task_14", "Task_2")
add_edge("Task_20", "Task_2")

result = build_workflow("pmo-17")  # PASS ✓ → Stage-4 complete
```

Figure 9. Attempt 3 code — condensed excerpt. The agent re-emits the full 43 nodes and 34 edges. The two "key fix" blocks show the merge-node removal and direct-feeder wiring that resolved the deadlock.

Output field semantics:

`failure_stage` One of `deploy`, `activation`, `execution`, `logical`, or `none`. Lifecycle phase at which the workflow failed, or `none` on success.

`failure_source` One of `translation_error`, `bpmn_quality`, `n8n_limitation`, or `unknown`. Root-cause category.

`is_fixable` Boolean — whether re-running the Translation Agent with this feedback plausibly fixes the issue.

`root_cause` One-sentence explanation of the failure.

`bpmn_construct_responsible` Name of the BPMN construct (e.g. "parallel gateway", "intermediate event") responsible, or `none`.

`suggested_fix` A concrete, actionable suggestion — the translator reads this field first on retry.

`severity` One of `blocking`, `degraded`, or `cosmetic`. Impact classification.

`confidence` One of `high`, `medium`, or `low`. Diagnostic confidence.

H. Reproducibility

H.1. Environment

Python $\geq$ 3.12, $<$ 3.13. Install with `uv`:

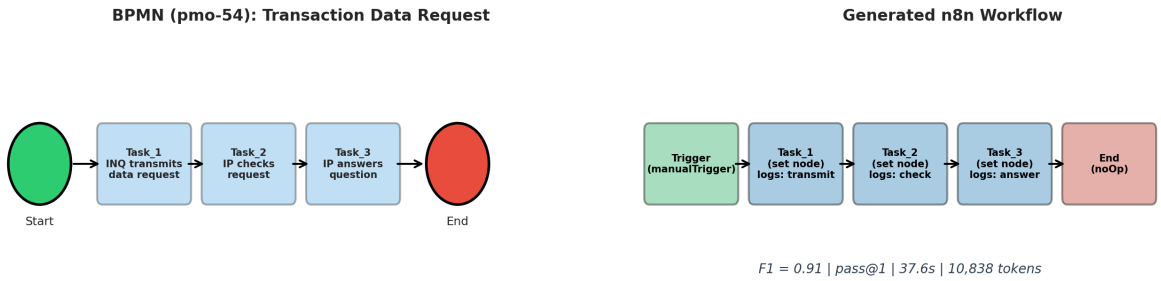


Figure 10. BPMN element → n8n node mapping (nine-node allowlist). Every element type appearing in the PMo corpus maps to exactly one n8n node type under the rules described in Section 3.2.2.

```
cd src/automation
uv sync                           # creates .venv
cp .env.example .env              # fill in secrets
uv run python eval.py [flags]     # curated track
uv run python ../end_to_end.py [flags]    # end-to-end
```

Required environment variables (names only — set in `src/automation/.env`):

`FPT_AI_API_KEY` Bearer token for FPT AI Factory serverless inference.

`FPT_AI_BASE_URL` FPT endpoint, default `https://mkp-api.fptcloud.com`.

`FPT_AI_MODEL` Model identifier for the Translation Agent (e.g. `gemma-4-31B-it` on the curated track; `Qwen3-32B` on the end-to-end track after Kimi-K2.5 was deprecated on FPT).

`FPT_AI_FEEDBACK_MODEL` Model identifier for the Feedback Agent (same conventions as above).

`N8N_BASE_URL` Local n8n REST base, default `http://localhost:5678`.

`N8N_API_KEY` n8n API key (Settings → API in the n8n UI).

H.2. Curated Gemma-4 track (Translation Agent on ground-truth PMEs)

Section 6.4's numbers come from this command, which bypasses Stages 1–3 (message store, discovery, BPMN generation) and runs Stages 4–5 (PME→n8n + simulate+feedback) on the 55 PMo ground-truth PMEs:

```
cd src/automation
uv run python ../end_to_end.py \
  --start-stage 4 \
  --pme-dir     $REPO/data/pmo-dataset/pme \
  --n8n-dir     $REPO/data/n8n_workflows/generated_gemma4 \
  --results-dir $REPO/results/gemma4_curated \
  --max-retries 2 \
  --skip-existing
```

With `FPT_AI_MODEL=gemma-4-31B-it` and `FPT_AI_FEEDBACK_MODEL=gemma-4-31B-it` in `.env`. Stage 4 (PME→n8n translation on 55 PMEs with 16 parallel workers) wall time: 1476 s (∼25 min). Stage 5 (deploy+execute+feedback, sequential per PME) wall time: ∼5865 s (∼1.6 h). Total pipeline: ∼2.1 h.

H.3. End-to-end track (all five stages chained)

```
python src/end_to_end.py \
  --transcripts-dir data/transcripts \
```

```
--store-dir       data/end_to_end/message_store \
--bpmn-dir        data/end_to_end/bpmn \
--pme-dir         data/end_to_end/pme \
--n8n-dir         data/end_to_end/n8n_workflows \
--results-dir     data/end_to_end/simulation_results \
--provider        gemini \
--max-retries     1
```

Stages: (1) `prepare_store.py` flattens annotated transcripts into a message store; (2) `react_agent.py` runs ReAct discovery over the store (provider `gemini` or `anthropic`); (3) `bpmn_to_pme.py` converts each generated BPMN XML to PME JSON; (4) `agent.py` translates each PME to n8n JSON (FPT model per `.env`); (5) `pipeline.py` deploys, executes, trace-validates, and runs the feedback loop.

H.4. Output CSV: `pipeline_summary.csv`

Each row is one PMo model; 14 columns:

`model_id` Two-digit PMo model id (`01`–`55`).

`num_elements` Total count of tasks + events + gateways in the ground-truth PME.

`num_tasks` Task count.

`num_gateways` Gateway count (exclusive, parallel, inclusive, event-based).

`num_seq_flows` Sequence-flow edge count.

`regen_success` Boolean: Stage-4 translation succeeded within the retry budget.

`regen_pass_at` Attempt number on which Stage-4 first passed (1, 2, or 3 for pass@3), or empty if `regen_success=False`.

`regen_attempts` Total Stage-4 attempts used.

`regen_tokens` Cumulative tokens across all Stage-4 attempts (may be empty when not recorded).

`pipeline_status` One of `pass` (Stage-4 passed and Stage-5 trace F1 ≥ 0.8), `fail` (Stage-4 passed but Stage-5 failed or F1 < 0.8), or `regen_fail` (Stage-4 never passed).

`f1` Trace-replay F1 (0.0–1.0); empty if Stage-5 did not execute.

`failure_stage` Earliest Stage-5 failure: `deploy`, `activation`, `execution`, `logical`, or empty on pass.

`failure_source` Feedback-Agent root-cause category: `translation_error`, `bpmn_quality`, `n8n_limitation`, `unknown`, or `none`.

`missing_nodes` Pipe-separated list of BPMN element IDs absent from the executed trace.