Architecture & Initial Steps
Bernardo's Working Notes

BMW × MIT GenAI Lab
Continual Learning Agents Research

Working directory: `Experimentation/Bernardo`.
Guided by: Project Plan - Draft.pdf and GEPA_paper.pdf.

Approach: Extra granular, narrative-first. More English, less building for now. Focus on *how* we would approach architecture, data, and evaluation—and on showing **improvement attributable to better prompts**, not raw document-understanding accuracy.

Proposed stack: Trace (execution + tracing + feedback) + **GEPA** (reflective prompt optimizer). Trace runs the agent and records what happened; GEPA uses that record plus feedback to propose a better prompt.

---

1 High-Level Flow of the Project

End-to-end, the pipeline looks like this. Each line is one "layer"; the flow is top to bottom.

1. **PDF in.** We have a document (e.g. a repair order). It may be image-based (no extractable text).
2. **Document → representation (fixed tools).** We use **fixed** tools—e.g. a **vision model** (render PDF pages to images) or **OCR** (extract text)—to turn the PDF into something the LLM can read. We do *not* optimize these tools; the same conversion is used for baseline and every optimization step.
3. **Agent = LLM + system prompt.** The "agent" is: we send to the LLM (a) a **system prompt** (the only thing we will optimize) and (b) the **document representation** from step 2. The LLM returns an interpretation (e.g. extracted fields in JSON). So: **agent(prompt, document_representation) → structured output.**
4. **Trace: we put the agent in a graph; the prompt is a trainable node.** We implement the agent inside **Trace**. The system prompt is stored in a **Trace trainable node** (e.g. `node(..., trainable=True)` holding the prompt text). When we run the agent, Trace **traces** execution: it records which prompt was used, the LLM call(s), and the output. So we get a **trace** (what was sent, what was returned). The "trace prompt node" is simply: *the place in Trace where the system prompt lives so it can be read by the LLM and later updated by the optimizer.*
5. **Evaluation: score + text feedback (μf).** We compare the agent's output to **ground truth** and compute a **score** (metric μ) and **text feedback** (e.g. "RO number wrong; VIN

missing"). That's our **feedback function** μ**f**. Trace can pass this feedback (and the trace) to the optimizer.

6. **GEPA proposes a new prompt.** We use **GEPA** as the optimizer (instead of Trace's built-in OptoPrime/TextGrad). GEPA takes the **trace** (what the agent did) and the **feedback** (score + text), and a **reflection LLM** proposes a **new system prompt** that should fix the observed mistakes. That's the "suggested reflection on the prompt."
7. **Update the prompt node and repeat.** We write GEPA's proposed prompt into the **same Trace trainable node**. Then we run the agent again (same or next document), get a new trace and feedback, and repeat. So: **run agent (Trace) $\rightarrow$ μf $\rightarrow$ GEPA reflects and suggests new prompt $\rightarrow$ update node $\rightarrow$ run again.**
8. **Baseline vs optimized.** For the **baseline**, we run the agent once with an initial hand-written prompt (no GEPA steps) and record the score. For **optimized**, we run the GEPA loop for N iterations, then record the score with the final prompt. The **delta** is the improvement we report as "from better prompts."

In one sentence: We build an agent (LLM + prompt) that uses fixed tools (vision/OCR) to read the document; we run it inside Trace so the prompt is a trainable node and execution is traced; we evaluate with μf (score + text); GEPA uses the trace and feedback to suggest a new prompt; we update the node and repeat until we have a better prompt to report.

---

2 What Stakeholders Care About (And What We Optimize For)

Their priority: They want to see **improvement that is clearly attributable to better prompts**—not to a bigger model, more training data, or fancier document parsing. The story is: *"We took the same LLM and the same documents; we only changed the system prompt, and the system got better."* Absolute accuracy (e.g. "we extract 95% of fields correctly") is secondary. What matters is the **delta**: baseline prompt vs evolved prompt, same model, same eval set.

Implication for design: We must be able to hold everything constant (model, data, evaluation procedure) and change *only* the prompt. Then we measure the same metric before and after. Any gain is, by construction, prompt-attributable.

End goal in plain English: Take a general PDF that fits a schema (like the repair-order schema they provided), feed it to an LLM, and have that LLM interpret it **better than a baseline LLM would**—where "better" is measured consistently, and the only lever we pull is the system prompt. So: same model, same documents, better instructions $\rightarrow$ better interpretation.

3 End-to-End Story We Are Telling

1. We define a **task**: e.g. "Given this PDF (and its schema), extract or reason about these fields."
2. We build a **baseline**: one (or a few) system prompt(s), written in plain English, that describe the task and the document structure. We run the **same** LLM on the **same** evaluation documents with this baseline prompt and record scores.
3. We run a **prompt-optimization loop**: an automated process that proposes changes to the prompt(s), re-runs the system on the same eval set, and keeps changes that improve the score. The model and the eval set do not change; only the prompts do.

4. We **compare**: baseline prompt vs optimized prompt(s) on the same metric and the same documents. The improvement is our main result—*"prompt evolution yielded a +X% gain over baseline."*

So the architecture, data flow, and evaluation design all need to support this story: **controlled experiment, single variable (prompt), measurable delta.**

4 Trace + GEPA Framework (Proposed)

We propose using **Trace** and **GEPA** together: Trace provides the execution and tracing layer; GEPA provides the prompt-optimization logic. They connect at "run agent → get trace + feedback → optimizer updates prompt."

4.1 What Trace Does

- **Execution:** We implement the agent (LLM + prompt, fed with document representation) as a Trace **computation graph**. The system prompt is a **trainable node** (e.g. `node("...", trainable=True)` or equivalent). Tools like vision or OCR run *outside* or *before* the graph: they produce the document representation that we feed in as a non-trainable input.
- **Tracing:** When we run the graph, Trace records the **trace**: which nodes ran, what values (e.g. prompt text, LLM input/output) they had. So we have a structured record of "what the agent did."
- **Feedback:** We attach our **feedback function** μ**f** (score + text) to the output. Trace can pass the trace and the feedback to an optimizer.

So: **Trace = where the agent runs and where the prompt lives (as a node), and how we capture what happened.**

4.2 What GEPA Does

- **Input:** The **trace** (from Trace) and the **feedback** (score + text from μf).
- **Process:** A **reflection LLM** reads the trace and feedback and proposes a **new system prompt** that should address the observed failures (e.g. "emphasize checking the RO number in the header").
- **Output:** The new prompt text. We write it into the Trace trainable node and run again.

So: **GEPA = the optimizer that suggests prompt changes from trace + feedback.**

4.3 How They Connect

1. **Build the agent in Trace:** One (or more) trainable node(s) hold the system prompt; the graph calls the LLM with (prompt, document_representation) and returns structured output.
2. **Run the agent** on a document (or minibatch). Trace produces a **trace**.

3. **Evaluate:** Run μf(agent_output, ground_truth) $\rightarrow$ (score, feedback_text).
4. **Call GEPA:** Pass (current prompt, trace, score, feedback_text). GEPA returns a **proposed new prompt**.
5. **Update:** Set the Trace trainable node to the new prompt. Go to step 2 (next iteration or next document).

Trace does not need to "know" GEPA; we just use GEPA's output to update the node. So we can implement GEPA as a **custom optimizer** that, given Trace's trace and our μf result, returns the new prompt string, and we write that into the node. Alternatively, we run GEPA's loop externally and only use Trace for execution and tracing—either way, the flow is the same.

4.4 Why This Pair

- **Trace** gives us a clean way to (a) represent the prompt as a single updatable object, (b) trace execution for the optimizer, and (c) plug in different optimizers. We avoid building the "run agent $\rightarrow$ collect feedback" harness from scratch.
- **GEPA** gives us sample-efficient, reflection-based prompt updates that use natural-language feedback (e.g. "RO number wrong")—aligned with our μf and with stakeholder interest in interpretable improvements.

Together: **Trace = skeleton and tracing; GEPA = the brain that improves the prompt.**

5 Architecture — How We Approach It (Granular)

5.1 What We Mean by "Architecture"

Here, *architecture* means: (a) how many "steps" or "modules" the system has, (b) what each step does, (c) where the **prompt** lives, and (d) how data flows from input (PDF) to output (interpretation). We keep it simple so that when we report "better prompts $\rightarrow$ better results," there is no ambiguity about what changed.

5.2 The Simplest Picture: One Module, One Prompt

The most conservative design is **one LLM call with one system prompt**. Input: the document (as images or as extracted text, depending on what we decide) plus the task description. Output: the interpretation (e.g. structured fields, or answers to questions). The **only** thing we optimize is that single system prompt. Any improvement in the metric is then unambiguously "better prompt."

Why start with one module: So we don't have to argue about which module's prompt caused the gain. One prompt, one delta. Later we can split into multiple modules (e.g. "describe document" then "fill fields") if we want to study *which* part of the instruction benefits most from optimization—but that's a second phase.

5.3 Where the Prompt Lives and What It Contains

The prompt is the **system (or instruction) text** we send to the LLM before we send the actual document. It should include:

- **What the document is:** e.g. a repair order with sections (ASI, BWO, CSI, JSI, WSI, ISI) and common header fields (dealer, vehicle, dates, RO number, VIN, etc.), as in the schema they provided.
- **What we want the LLM to do:** e.g. "Extract the following fields from the header," or "Answer the following question about this document."
- **How we want the output:** e.g. JSON with specific keys, or short answers.

We do *not* change the model, the document encoding, or the evaluation logic when we "optimize"; we only change this text. So the architecture must clearly separate "prompt text" from "model" and "evaluation."

5.4 Optional: Multiple Modules (Later)

If we later split into multiple steps (e.g. "first describe each section in natural language, then fill a structured form from that description"), each step can have its own prompt. Then we have multiple prompts to optimize. GEPA-style methods update one module at a time and use feedback to decide *which* module to change. For the first pass, though, a single module keeps the narrative simple: one prompt, one improvement curve.

5.5 Control Flow in Words

- **Input:** A PDF (or a set of page images) and, optionally, a task (e.g. "extract these fields" or "answer this question").
- **Step 1:** We convert the PDF into whatever representation the LLM accepts (e.g. images per page, or OCR'd text). This conversion is **fixed**—we do not optimize it. Same conversion for baseline and optimized runs.
- **Step 2:** We send to the LLM: (system prompt + document representation + optional user query). One or more calls, depending on single- vs multi-module.
- **Step 3:** We take the LLM output and parse it into our output schema (e.g. a dict of fields). Parsing rules are **fixed** so that, again, the only variable is the prompt.
- **Output:** Structured interpretation (e.g. field values, or answers).

So: **data in → fixed encoding → LLM(prompt, encoded doc) → fixed parsing → data out.** The only thing we vary is the content of the prompt.

5.6 Why This Supports "Improvement from Prompts"

Because we fix the model, the document encoding, and the output parser, any change in the evaluation score when we swap prompts is due to the prompt. We are not improving document understanding by adding a better OCR or a bigger model; we are improving it by giving the same model better instructions. That is exactly what stakeholders want to see.

6 Data — How We Feed It (Granular)

6.1 What Data We Have

- **Schema / spec:** `overview.pdf` describes the repair-order document type—section types (ASI, BWO, CSI, JSI, WSI, ISI) and common header fields. This is our **reference for what "correct" looks like** and what we should ask the LLM to produce.
- **Sample documents:** Several PDFs (e.g. 201414, 678856, . . .) that are real (or realistic) repair orders. They appear to be image-based (no extractable text), so the LLM will need to work from **images** (or we add a fixed OCR step and feed text; either way, we keep that choice fixed for baseline and optimized runs).

6.2 How We Use the Schema

The schema tells us (a) what concepts exist in the document (sections, header fields), and (b) what we might ask the agent to extract or reason about. We **feed** the schema (or a concise summary of it) into the **system prompt** so the LLM knows the structure. We do not change the schema between baseline and optimized runs; we only change how we *instruct* the LLM to use that schema (wording, emphasis, examples if any). So the schema is part of the "fixed" context; the optimizable part is the rest of the prompt (task description, format, reasoning hints).

6.3 How We Feed the Document

- **Option A (vision):** Send page images to a vision-capable LLM. Same pages, same order, for every run. No change between baseline and optimized prompt.
- **Option B (OCR):** Run a fixed OCR pipeline on each PDF, get text (or structured blocks), and send that text to the LLM. Again, same OCR for everyone; we only change the prompt.

We pick one and stick to it. The point is: **document → same representation every time** so that the only variable is the prompt.

6.4 What We Need for Evaluation (Ground Truth)

To compute "did the LLM interpret this correctly?" we need **reference answers** for at least a subset of documents. We create these by hand: for each eval document, we write down the "correct" fields or answers (e.g. RO number, VIN, open date). This is our ground truth. We do *not* use it to train the model; we use it only to score the LLM's output. So:

- **Training/optimization set (for the prompt optimizer):** Documents + ground truth. The optimizer runs the system, sees the score and feedback, and proposes prompt changes. It may use a subset of documents to save cost.
- **Evaluation set:** Same or a held-out set of documents + ground truth. We report baseline score and optimized score on this set. If we want to be strict, we hold out some documents so the optimizer never sees them—then "optimized prompt" is evaluated on truly unseen documents. That would make the "improvement from prompts" claim even stronger.

So we feed: (1) schema (in the prompt), (2) document (fixed encoding), (3) optional explicit task (e.g. "extract these fields"). Ground truth is used only by the evaluation and feedback logic, not as part of the prompt at inference time (unless we explicitly design a few-shot setup; that would be a separate choice).

6.5 Summary of Data Flow

- **Into the prompt:** Schema summary + task description + (if we want) format instructions. All of this is the "prompt" we optimize.
- **Into the model (as content):** The document in a fixed representation (images or OCR text). Same for baseline and optimized.
- **Into the evaluator:** Model output + ground truth for each eval document. The evaluator returns a number (and optionally text feedback for the optimizer). No data "leak" from ground truth into the prompt.

7 Evaluation Metrics — How We Introduce Them (Granular)

7.1 Why Metrics Matter Here

We need a **single, repeatable way** to say "this run was better than that run." The metric is the yardstick. We use it (a) to decide whether a candidate prompt is an improvement (during optimization), and (b) to report "baseline vs optimized" at the end. If the metric is not aligned with "better interpretation," we might optimize for the wrong thing. So we choose metrics that reflect "did the LLM understand and use the document correctly?" in a way that stakeholders recognize.

7.2 What We Actually Care About Measuring

We said they care more about **prompt-attributable improvement** than about **absolute accuracy**. So we need:

1. **A metric that is sensitive to prompt quality.** If we change the prompt and the LLM does better on the task, the metric should go up. So the metric must depend on the LLM's output (e.g. extracted fields, answers) and on ground truth.
2. **A way to isolate the effect of the prompt.** We already did that by fixing model and data; the metric just needs to be computed the same way for baseline and optimized runs.
3. **Optional: interpretable feedback.** For a GEPA-style loop, we want not only a number but also **text feedback** (e.g. "RO number was wrong; VIN missing") so the optimizer can propose targeted prompt edits. So we may have: **numeric score** (for "better or worse") and **feedback text** (for "what to fix").

7.3 Order of Introduction: Metric First, Then Feedback

Step 1 — Define the task and the output schema.
We decide exactly what the LLM must produce (e.g. a list of field names and values). That defines "correct" vs "wrong" per field or per document.

Step 2 — Define a numeric metric μ.
Examples: (a) exact-match rate per field, (b) F1 over fields, (c) proportion of documents with all key fields correct, (d) a simple rubric (e.g. 0/1/2 points per field). We pick one (or a small set) and **stick to it** for both baseline and optimized runs. We document clearly: "We measure X; higher is better."

Step 3 — Implement μ over (model output, ground truth).
We write a function that, given the parsed model output and the ground truth for one document, returns a score. We run it on every eval document and aggregate (e.g. average). No prompt content here—just comparison of output to reference.

Step 4 — Extend to a feedback function μf (for optimization).
If we use a reflective optimizer (GEPA-style), we need more than a number: we need **why** the score was low or high. So we define μf that returns (score, feedback_text). For example: "Score 0.6. Missing: VIN. Wrong: RO number (expected 201414, got 201415)." The optimizer uses this text to propose prompt changes. The important part: μf is derived from the same notion of "correct" as μ; we don't change the definition of correct, we just add a textual breakdown.

7.4 Ensuring Improvement Is Attributable to Prompts

- **Same metric** for baseline and optimized: we report the same μ (e.g. average field accuracy) for both.
- **Same eval set:** same documents, same ground truth. If we use a held-out set for final reporting, we say so.
- **Same model and same document encoding:** no change between runs except the prompt.
- **Same parsing of model output:** we don't "fix" the output post hoc for one run and not the other. We parse the raw output the same way and then apply μ.

Under those conditions, any gain in the metric is attributable to the prompt. We can say: "With the baseline prompt we got X; after prompt optimization we got Y; the gain $(Y - X)$ is due to better prompts."

7.5 What We Report to Stakeholders

- **Baseline score:** metric μ on the eval set with the initial, hand-written prompt(s).
- **Optimized score:** metric μ on the same set with the evolved prompt(s).
- **Delta:** improvement (absolute or relative). We emphasize: "This gain comes from prompt optimization only; same model, same data."
- Optionally: a few examples where the optimized prompt fixed a mistake the baseline made, to make the story concrete.

We do *not* need to claim "we achieved 95% accuracy." We claim "we improved over the baseline by Z% by evolving the prompt."

8 Tying It Together: The Experiment in Plain English

1. **Define the task** using their schema (e.g. repair orders): what the LLM must output (fields or answers), and in what format.
2. **Freeze everything except the prompt:** one model, one way of feeding the PDF (vision or OCR), one output parser, one evaluation metric.
3. **Baseline:** write an initial system prompt that describes the schema and the task. Run on the eval set. Record the metric. This is "baseline LLM" in their terms—same model, naive instructions.
4. **Optimization loop:** automatically propose prompt changes (e.g. via reflection over failures), re-run on the same (or a subset of) documents, keep changes that improve the metric. Use the same μ (and optionally μf) throughout.
5. **Final comparison:** run the best-evolved prompt on the eval set. Report baseline vs optimized metric. The difference is the "improvement attributed to better prompts." The end goal—interpreting a general PDF (like their schema) better than a baseline LLM—is demonstrated by that improved metric, with the only change being the prompt.

9 Initial Steps — Narrative Checklist (No Code Yet)

- ☐ **Schema:** Read and summarize the overview/schema (sections, fields). Write a short "document spec" we can paste into the system prompt.
- ☐ **Samples:** Inspect 1–2 sample PDFs. Confirm they match the schema and note how we will feed them (images vs OCR). Decide and document the choice.
- ☐ **Task:** Write one paragraph: "The agent's job is to . . . Output format: . . . " This is the task we will score.
- ☐ **Output schema:** List the exact keys (or question IDs) we will extract. This defines "ground truth" and "model output" shape.
- ☐ **Ground truth:** Manually fill 5–10 eval documents with correct values. Store them in a fixed format (e.g. one JSON per document).
- ☐ **Metric μ:** Choose one primary metric (e.g. field-level exact match rate). Write in English how it is computed. Later we implement it.
- ☐ **Feedback μf (if needed):** Describe in English what text we want back when the model fails (e.g. "list wrong/missing fields"). This will drive prompt updates.
- ☐ **Baseline prompt:** Write the first version of the system prompt (schema + task + format). This is what we will compare against.
- ☐ **Architecture doc:** One page: "We have one module. Input is . . . Output is . . . The only thing we change is the prompt. Metric is . . . " So the whole experiment is reproducible on paper.

Once these are done, we have a thorough, granular plan. Building (data loading, API calls, eval script, optimizer loop) can follow this plan step by step.

---

10 References

- **Project Plan - Draft.pdf** — scope (no RL, no retraining), tools, milestones.
- **GEPA_paper.pdf** — compound system, reflective prompt evolution, Pareto selection, feedback function μf.
- **Pin — Trace as skeleton:** `PIN_TRACE_SKELETON.md` (Microsoft Trace: https://github.com/microsoft/Trace — PyTorch-like optimization loop for agents; possible implementation skeleton.)
- GEPA code: https://github.com/gepa-ai/gepa