Automaton Auditor

Architecture & Decisions — Digital Courtroom Audit System

🔹 Architecture Overview (Two Pillars)

🔷 Parallel Evidence Collection (Detective Layer)

After loading the rubric and routing (context_builder), evidence gathering **fans out** so that multiple detectives run **in parallel** on the same inputs: RepoInvestigator (clone, AST, git history), DocAnalyst (PDF report), and VisionInspector (diagrams). PDF is converted once (`pdf_preprocess`), then **doc_detective** and **vision_detective** run in parallel. All detective outputs **fan in** at **evidence_aggregator**, where state reducers merge evidence by source (`repo`, `docs`, `vision`). No detective sees another's output; they produce objective, non-opinionated evidence only.

🔷 Judge System (Dialectical Bench)

After evidence is aggregated, the **Dialectical Bench** evaluates it via a **second fan-out/fan-in pattern**: three distinct nodes— prosecutor_node (strict, gap-focused), defense_node (charitable, effort-focused), and tech_lead_node (pragmatic, architecture-focused)—run **in parallel** via `Send()`. Each node produces a **JudicialOpinion** (score, argument, cited evidence) **per rubric criterion**. All opinions **fan in** at **judges_aggregator** (state merged by `operator.add` on `opinions`).

Theoretical Depth (Rubric Alignment)

- **Fan-In / Fan-Out:** Tied to specific graph edges:
  - Fan-out #1 (Detectives): `context_builder` → conditional edges (`Send`) to `repo_detective` and `pdf_preprocess`; `pdf_preprocess` → `Send` to `doc_detective` and `vision_detective`.
  - Fan-in #1: `repo_detective`, `doc_detective`, `vision_detective` → `evidence_aggregator` (state merged by `operator.ior` on `evidences` dict).
  - Fan-out #2 (Judges): `evidence_aggregator` → `judges_router` returns `[Send("prosecutor_node", state), Send("defense_node", state), Send("tech_lead_node", state)]`.
  - Fan-in #2: `prosecutor_node`, `defense_node`, `tech_lead_node` → `judges_aggregator` (state merged by `operator.add` on `opinions` list).
- **Dialectical Synthesis:** Three parallel judge personas (Prosecutor, Defense, Tech Lead) evaluate the same evidence with actively conflicting philosophies. The Chief Justice applies deterministic rules (Rule of Security, Rule of Evidence, functionality weight, variance re-evaluation) to produce a single verdict per criterion.
- **Metacognition:** The Chief Justice performs metacognitive re-evaluation at multiple levels:
  1. **Fact supremacy:** Detective evidence overrides judge opinions.
  2. **Variance re-evaluation:** High variance triggers downgrade (PASS → PARTIAL).
  3. **Security override:** Security vulnerability caps score.
  4. **Dissent summary:** Explicit explanation of internal disagreement.
- **State Synchronization:** Annotated reducers on `AgentState` ensure parallel nodes write without race conditions.

◆ 1. Architecture Decisions

1.1 Pydantic Over Dicts

Decision: All evidence, judicial, and report payloads use **Pydantic `BaseModel`**; graph state uses **`TypedDict`** with **`Annotated`** **reducers** for merge semantics.

Rationale:

- **Structured output enforcement:** Judges must return a `JudicialOpinion` (score, argument, cited_evidence). Using `.with_structured_output(JudicialOpinion)` guarantees the LLM output is validated and parseable; freeform dicts would require brittle parsing and retries.
- **Intent–code traceability:** The rubric requires "Pydantic and Annotated reducers." Typed models make it explicit what each node reads/writes and satisfy the "State Management Rigor" and "Structured Output" criteria.
- **Reducer safety:** `AgentState` uses `Annotated[dict[str, list[Evidence]], operator.ior]` and `Annotated[list[JudicialOpinion], operator.add]`. Reducers are defined once on the type; parallel nodes (detectives, judges) merge without overwriting.

Where it's used:

- `src/state.py`: `Evidence`, `JudicialOpinion`, `CriterionResult`, `AuditReport` (BaseModel); `AgentState` (TypedDict + Annotated reducers).
- `src/tools/repo_tools.py`: `CommitRecord`, `GitHistoryResult`, `GraphStructureResult` for AST/git outputs.
- Judge nodes: `JudicialOpinion` as the only allowed LLM response shape.

1.2 AST Parsing Strategy (No Regex for Graph/State)

Decision: Graph and state structure are inferred using **Python's `ast` module** (parse → walk → detect calls and names). No regex on source code for structure.

Rationale:

- **Reliability:** Regex breaks on formatting, comments, or multi-line calls. AST is the same regardless of style and gives real syntax (calls, args, names).
- **Rubric alignment:** The rubric requires verifying "builder.add_edge()" and fan-out/fan-in; that implies call-site analysis, not string search.
- **Maintainability:** One `analyze_graph_structure(path)` returns a `GraphStructureResult` (has_state_graph, has_add_edge, has_fan_out, has_evidence_aggregator, has_parallel_judges, node_names, edge_count).

How it's structured:

1. **Locate graph file:** `_ast_find_graph_file(repo_path)` checks `src/graph.py` then `graph.py`.

2. **Parse:** `ast.parse(source)`; on `SyntaxError` return a failed `GraphStructureResult` with details.
3. **Single walk:** `ast.walk(tree)` over all nodes:
   - **Calls:** Detect `StateGraph` (name containing "StateGraph"), `add_edge` (collect `(from, to)` from args), `add_node` (collect node name from first constant arg).
   - **Names:** Detect "EvidenceAggregator" and judge-related names (e.g. "judge" in id or "Judge").
4. **Derived flags:**
   - **Fan-out:** Any node with more than one outgoing `add_edge` (from_groups).
   - **Parallel judges:** At least two judge-related node names or names containing "judge".

1.3 Sandboxing Strategy (Clone & Tools)

Decision: All repo and system interaction is **sandboxed**: clone only into a **temporary directory**, **no `os.system`**, **subprocess with list args and full error handling**, **cleanup on failure**, **URL validation and remote verification**.

Concern	Approach
Where to clone	`tempfile.mkdtemp(prefix="repo_tools_")` → `parent / "repo"`. Never cwd or project root.
Shell / injection	No `os.system`. `subprocess.run(["git", "clone", "--quiet", repo_url, str(clone_into)], ...)` with list args.
Errors	Capture stdout/stderr, check returncode. On non-zero or timeout: `shutil.rmtree(cleanup_path)`, raise `CloneError`.
Timeout	120s for clone; 30s for `git log`; 10s for `remote get-url`.
Validation	CLI: `validate_github_url(repo_url)` before graph run. After clone: `_get_remote_origin` vs `_normalize_github_url(repo_url)`; on mismatch, cleanup and raise `CloneError`.

◆ 2. Known Gaps and Concrete Plan

2.1 Judicial Layer

Current state:

- **One node per persona:** `prosecutor_node`, `defense_node`, `tech_lead_node` are separate graph nodes, each processing all rubric criteria independently.
- **True parallel execution:** The graph wires these three nodes via `Send()` fan-out from `evidence_aggregator`. LangGraph executes them concurrently.
- **Structured output:** Each judge returns `JudicialOpinion` via `.with_structured_output()`; retries on parse failure (up to 3 attempts).
- **Cite validation:** After each judge response, `cited_evidence` refs are validated against the actual evidence blob. Invalid refs are filtered.

Gaps:

1. No true parallel execution of judges: Today `run_judges` is a single node that loops over dimensions and judges sequentially. Latency is N_dimensions × 3 × LLM_call.
2. Persona drift: Prompts are static; long evidence or many criteria can dilute persona instructions.
3. Cite-only evidence: We don't validate that cited refs exist in the evidence blob.
4. Score scale: Judges use 0–10; report normalizes to 1–5.

Concrete plan:

#	Action	Owner / location
1	~~**Parallelize judge invocations:**~~ **DONE.** Three separate nodes wired via `Send()` fan-out.	`src/graph.py`, `src/nodes/judges.py`
2	~~**Stricter persona prompts:**~~ **DONE.** <50% overlap; explicit adversarial/charitable/pragmatic instructions.	`src/nodes/judges.py`

#	Action	Owner / location
3	~~**Cite validation:**~~ **DONE.** `_validate_cited_refs` filters invalid refs post-response.	`src/nodes/judges.py`
4	**1–5 in prompt:** Change judge prompt to "score (1–5)" and set Pydantic `score: int` with `ge=1, le=5`.	`src/nodes/judges.py`, `src/state.py`

2.2 Synthesis Engine (Chief Justice)

Current state:

- **Fully deterministic:** No LLM. Rules: Rule of Security, Rule of Evidence, functionality_weight, dissent_requirement, variance_re_evaluation.
- **Config from rubric:** `synthesis_rules` and `synthesis_config` loaded by context_builder.
- **Output:** `AuditReport` with executive_summary, criterion_breakdown, remediation_plan.

Gaps:

1. **Variance re-evaluation is score-only:** When variance is high we only downgrade by verdict band.
2. **Rule of Security is keyword-based:** No structured "security finding" from detectives.
3. **Remediation is template-based:** We don't generate step-by-step guidance from judge arguments.
4. **No explicit dissent narrative:** No single "Supreme Court reasoning" paragraph.

◆ 3. StateGraph Flow (Diagrams)

3.1 Parallel Evidence Collection (Detective Layer)

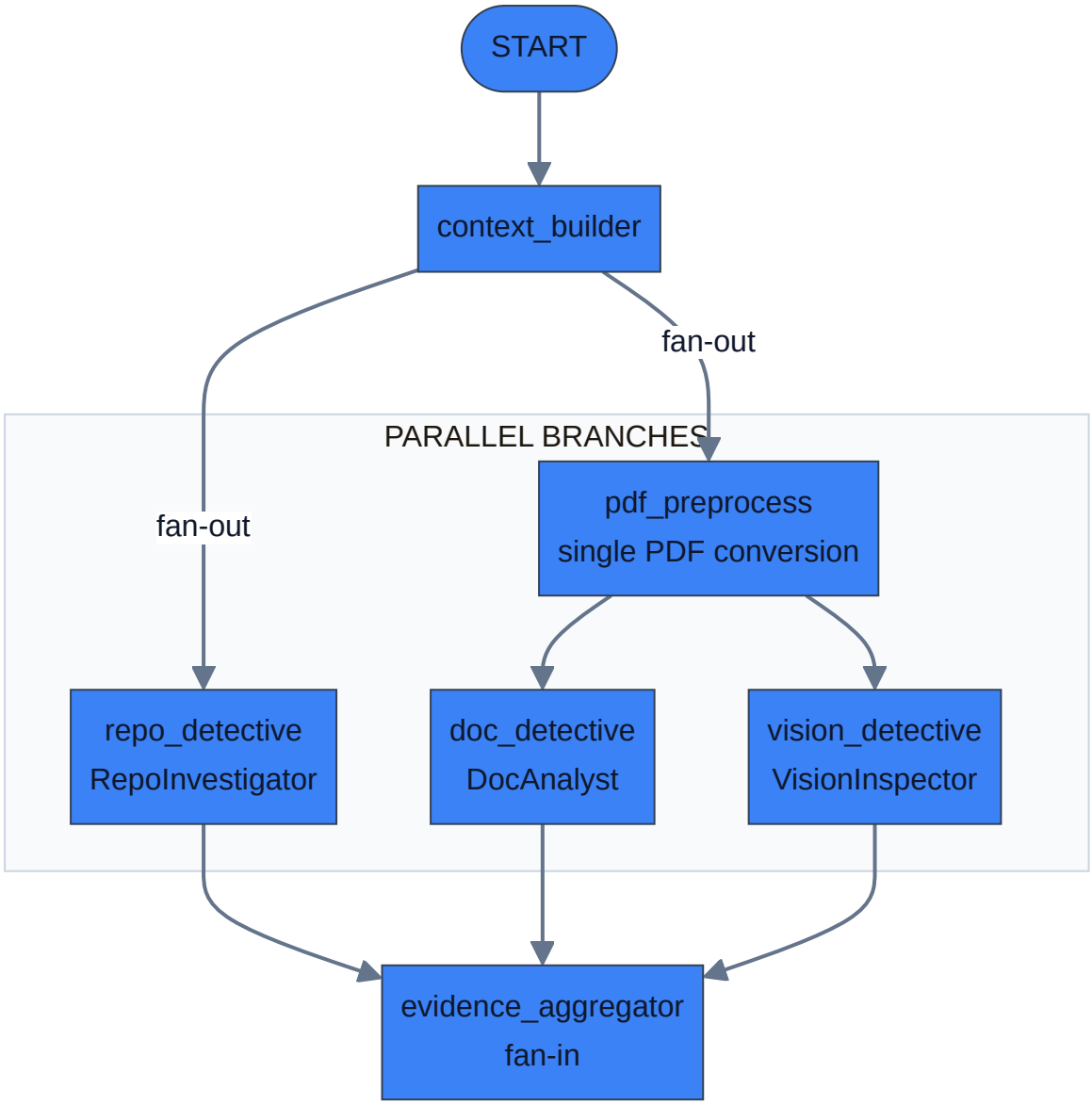

Figure 1: Parallel Evidence Collection - Detective Layer with explicit fan-out/fan-in

- **Fan-out:** `context_builder` conditionally sends to **repo_detective** and/or **pdf_preprocess** in parallel. After `pdf_preprocess`, **doc_detective** and **vision_detective** run in parallel.
- **Parallel branches:** RepoInvestigator, DocAnalyst, and VisionInspector write to different keys (`evidences["repo"]`, `evidences["docs"]`, `evidences["vision"]`).
- **Fan-in:** All branches converge at **evidence_aggregator**; state reducers (`operator.ior`) merge outputs.

3.2 Judge System (Dialectical Bench)

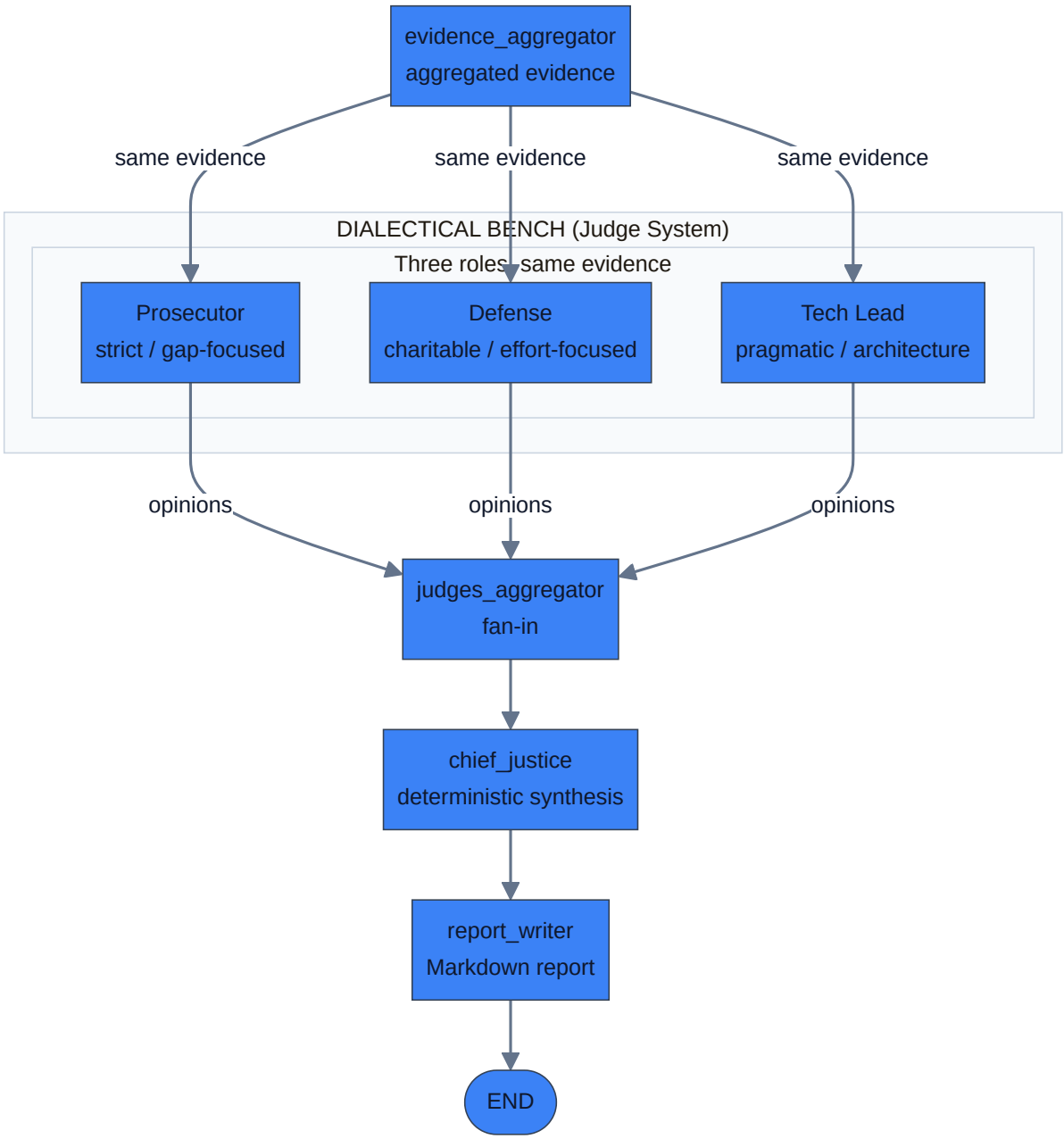

Figure 2: Dialectical Bench - Three parallel judges with fan-in at judges_aggregator

- **Same evidence:** Prosecutor, Defense, and Tech Lead all receive identical aggregated evidence and rubric.
- **Three roles:** Each role has distinct system prompt (Prosecutor: low scores for gaps; Defense: reward effort; Tech Lead: technical correctness).
- **Fan-in:** All opinions collected via state reducer `operator.add` on `opinions`.
- **Synthesis:** `chief_justice` applies deterministic rules to produce final verdict.

3.3 Full StateGraph (End-to-End)

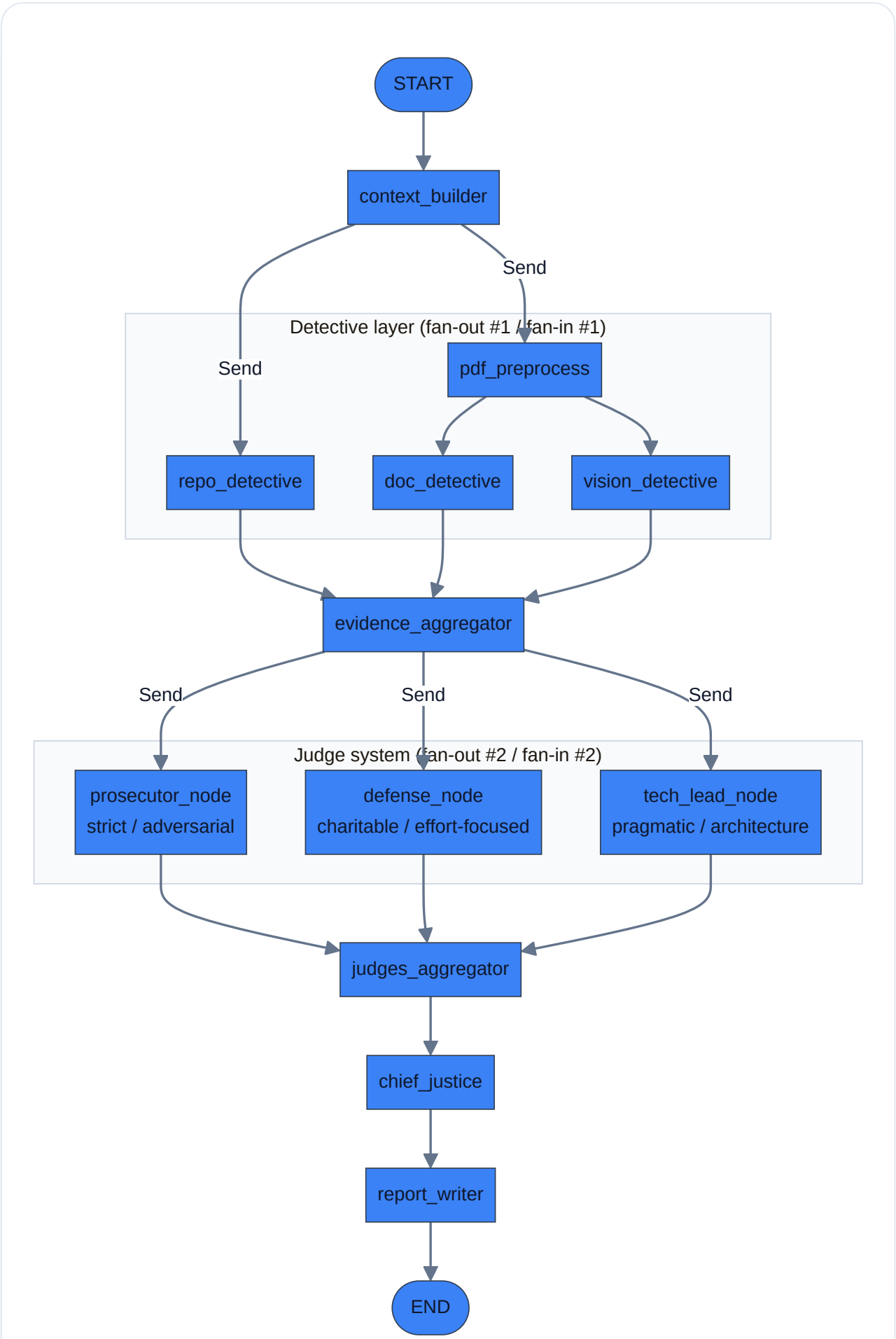

Figure 3: Full StateGraph - Two parallel fan-out/fan-in patterns highlighted

- **Fan-out #1 (Detectives):** `context_builder` → `Send(repo_detective)`, `Send(pdf_preprocess)` → `Send(doc_detective)`, `Send(vision_detective)`.
- **Fan-in #1:** All detectives → `evidence_aggregator`. State merged by `operator.ior` on `evidences`.
- **Fan-out #2 (Judges):** `evidence_aggregator` → `Send(prosecutor_node)`, `Send(defense_node)`, `Send(tech_lead_node)`.
- **Fan-in #2:** All judges → `judges_aggregator`. State merged by `operator.add` on `opinions`.

3.4 State Reducers (Where Parallel Writes Merge)

State key	Reducer	Role
`evidences`	`operator.ior`	Detectives write to different keys (`repo`, `docs`, `vision`); merge by key.
`opinions`	`operator.add`	Each judge call appends a list of opinions; concatenation preserves all.
`criterion_results`	`_last_wins`	Chief Justice writes once; last write wins (fan-in safety).
`final_report`	`_last_wins`	Same.

4. File Map (Implementation)

Area	Path
State & models	`src/state.py`
Graph definition	`src/graph.py`
Context & rubric	`src/nodes/context.py`
Detectives	`src/nodes/detectives.py`
Judges	`src/nodes/judges.py`
Chief Justice & synthesis	`src/nodes/justice.py`
Report Markdown	`src/report_serializer.py`
Repo tools (clone, AST, git)	`src/tools/repo_tools.py`
Doc/PDF tools	`src/tools/doc_tools.py`
Vision tools	`src/tools/vision_tools.py`
Rubric (dimensions, synthesis_rules)	`rubric/week2_rubric.json`

Last updated to match the implementation: single `run_judges` node per criterion, Pydantic/TypedDict state, AST-based graph analysis, and sandboxed clone with URL validation and remote check.