Etabot: Multi-Agent Verification Management with Chat-Accessible Midpoint Metrics and Finish-Date Forecasts

Zili Fang

zili.fang@silabs.com

Silicon Labs, 400 W Cesar Chavez St, Austin, TX 78701, USA

Abstract— Etabot is a local-first, multi-agent framework that automates verification management end to end—from running regressions and merging a unified coverage database to triaging failures, collecting midpoint metrics, forecasting finish dates, and producing manager-ready status. The system focuses on orchestration and data accessibility rather than bespoke tooling: A planner translates a short natural-language intent into a typed plan; a simple orchestrator executes agents in sequence; and midpoint data are exposed consistently to both a CLI and an MCP-enabled chatbot. The coverage pipeline collects code and functional coverage, parses text reports and appends design-only pass rate (infrastructure failures excluded) to a unified metrics file. The forecaster agent is designed to be reset-aware and replaceable so that coverage definition changes do not destabilize schedules and teams can adopt alternative models without changing the flow. All side-effectful steps are dry-run by default and the system can operate fully on-prem without data egress.

Index Terms— SystemVerilog/UVM; regression orchestration; coverage analytics; design-only pass rate; MCP; midpoint status; P50/P80 forecasts; local-first; agent flow

I. INTRODUCTION

Verification teams repeatedly answer two questions: what is the status, and when will we be done? Collecting logs, merging coverage, separating infrastructure issues from design failures, and summarizing for managers takes time away from engineering. Etabot turns these ad-hoc actions into a repeatable on-prem agent flow that is vendor-agnostic at the integration points and surfaces the same midpoint metrics through a CLI and an MCP-enabled chatbot. This paper describes the agent orchestration, how midpoint data are produced and presented. The implementation presented in this paper was adapted from vibe coding. All the codes were generated from a large language model and tested manually.

II. BACKGROUND AND RELATED WORK

Coverage data interchange and reporting are typically handled through simulators specific coverage databases and reporting utilities. Farm orchestration often relies on a farm scheduler (e.g., via a batch submission command). On the agent-framework side, a multi-agent framework provides primitives for building multi-agent flows with custom control and memory across sessions. For connecting assistants safely to on-prem resources, the Model Context Protocol (MCP) has emerged as an open standard [1, 2].

Throughout this work we report probabilistic finish dates using percentiles of a simulated completion-time distribution: P50 is the median (a "most likely" date) and P80 is the 80th percentile (a more conservative date). Reporting percentiles conveys uncertainty explicitly rather than a single point estimate, which is often more useful for planning and risk communication in verification programs.

A large body of recent work targets coverage closure with AI/ML. Industrial papers and DVCon proceedings report accelerating functional coverage via machine-learning-based selection or generation of tests, including reinforcement learning and iterative learning loops [3, 4, 5, 6, 7]. Academic and preprint efforts demonstrate ML pipelines that optimize regression effort for coverage goals (e.g., PyUVM + ML) [6]. These works focus on closing coverage; in contrast, Etabot focuses on managing verification operations and forecasting schedule reach to a coverage target while integrating pass-rate and bug

backlog. We orchestrate existing tools, capture midpoint metrics, and present explainable P50/P80 ranges as planning aids rather than optimizing stimulus selection or test scheduling.

III. SYSTEM OVERVIEW

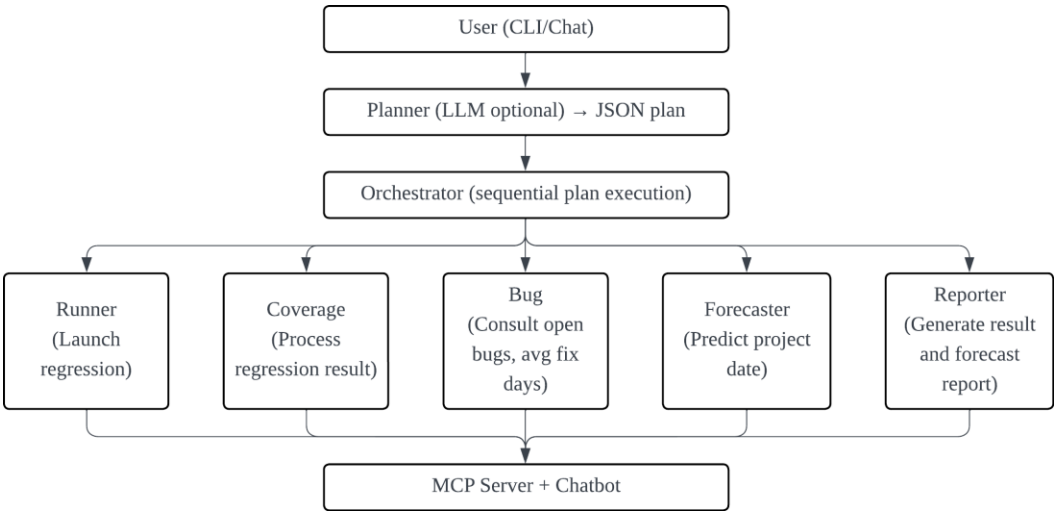

Fig. 1. Agent flow diagram

The user issues an intent (for example, “nightly regression, coverage, open bug, forecast 95%”) via CLI or chat. The planner returns a strict JSON plan. The orchestrator executes:

- Runner agent: renders/submits jobs to the farm (a farm scheduler and chosen simulator).
- Coverage agent: merges a unified coverage database and generates a text report; a parser extracts code and functional coverage and classifies failures to exclude infrastructure issues (license/timeout/OOM/filesystem/network/scheduler/tool-crash) and computes a design-only pass rate. See Appendix A METRICS_TS.CSV and Appendix B RESULTS.CSV as output examples
- Bug agent: queries an issue tracker for open-bug count and average fix time using JQL.
- Forecaster agent: consumes metrics (and optional bug stats) and outputs P50/P80 dates; the implementation is swappable.
- Reporter agent: writes a concise markdown including latest pass rate, coverage status, bug status, and forecast.

Outputs are also exposed through an MCP server so external assistants can call tools and read resources on-prem.

IV. IMPLEMENTATION (AGENTS AND ORCHESTRATION)

The implementation centers on a small set of replaceable agents wired by a simple orchestrator. Below we show key code that binds tasks to agents, turns a short intent into a sequential plan, executes that plan, collects midpoint metrics, and surfaces results to the CLI and chatbot. We emphasize the agent registry, planner, and orchestrator because they are the stable contracts and extension points that make the system portable across tools, reproducible under review, and safe to run on-prem. The registry makes side effects and integrations explicit and lets teams swap adapters without changing the flow. The planner shows how natural language becomes a deterministic plan—LLM-optional and rule-backed—so the same intent yields the same steps. The orchestrator exposes how data moves between tasks and how execution is gated (dry-run versus real), which underpins auditability and CI integration. Together these three pieces are the minimal surface a reader needs to extend the system, change vendors, or replace the forecasting model later without touching user workflows.

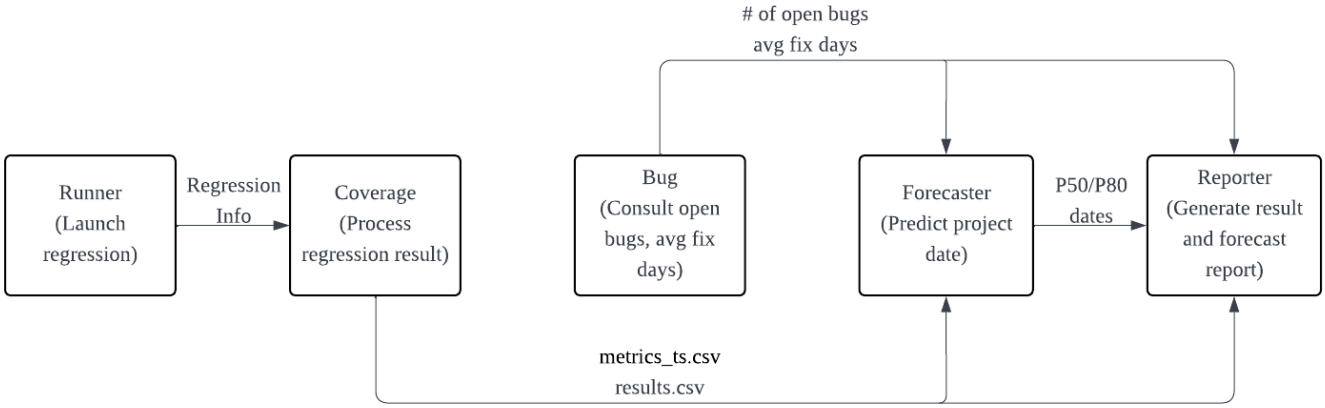

Fig. 2. Full plan execution diagram

The agent registry is a tiny directory that maps stable task names to the functions that implement them. The orchestrator never imports agents directly; it looks up a task by name and calls the bound function with a single parameters dictionary, which returns a small results dictionary. Because this binding sits in one place, you can swap adapters (simulator, scheduler, issue tracker, forecaster) without touching the orchestration code, and you can replace real agents with mocks for testing. All side-effecting agents honor a dry_run flag so commands can be rendered without executing.

```
# registry.py (excerpt)
from typing import Dict, Callable
Registry: Dict[str, Callable[[dict], dict]] = {}
def register(task: str):
def deco(fn: Callable[[dict], dict]):
Registry[task] = fn
return fn
return deco
@register("merge_coverage")
def _merge(p):
from .coverage_agent import merge_coverage
return merge_coverage(p)
@register("report_coverage")
def _report(p):
from .coverage_agent import report_coverage
return report_coverage(p)
@register("collect_metrics")
def _collect(p):
from .metrics_agent import collect_metrics
```

```
return collect_metrics(p)
@register("jira_fetch")
def _jira(p):
from .jira_agent import fetch_open_and_completed
return fetch_open_and_completed(p.get("open_jql"), p.get("completed_jql"),
p.get("base_url"), p.get("email"), p.get("api_token"))
@register("forecast_dual")
def _forecast_dual(p):
from .forecaster import forecast_dual_from_metrics
return forecast_dual_from_metrics(p["metrics_ts_csv"], p["code_target"], p["func_target"], p.get("slots", 64),
bugs_open=p.get("bugs_open", 0), avg_fix_days=p.get("avg_fix_days", 0.0))
@register("report")
def _reporter(p):
from .reporter import render_markdown
return {"report_md": render_markdown(p)}
```

Listing 1. Agent registry and task binding (replaceable adapters)

The planner turns a short request like “nightly regression, wait 8h, coverage, metrics, forecast 95%” into an ordered JSON plan. It recognizes common verbs, fills in sensible defaults, and emits steps in the right order—run, wait, merge, report, collect metrics, bug stats, forecast, report—marking side effects as dry-run unless execution is explicitly enabled. If a local or on-prem LLM is available it can be consulted to enrich or parameterize the plan; if not, the rule set handles the intent deterministically. The result is a clean, reproducible artifact that you can log, diff, and replay.

```
# planner.py (excerpt)
def plan_from_intent(intent: str) -> dict:
s = (intent or "").lower(); T = []
if "regression" in s or "nightly" in s:
T.append({"task":"run_regression","params":{"dry_run": True}})
if "coverage" in s:
T += [
{"task":"merge_coverage","params":{"ucdb_inputs":["runs/run1.a unified coverage database","runs/run2.a unified coverage database"],
"output_ucdb":"out/merged.a unified coverage database","dry_run": True}},
{"task":"report_coverage","params":{"ucdb_input":"out/merged.a unified coverage database",
"output_report":"out/cov_report.txt","dry_run": True}}
]
if "metrics" in s or "collect metrics" in s:
```

```
T.append({"task":"collect_metrics","params":{"report_path":"out/cov_report.txt",
"results_csv":"examples/results.csv",
"patterns_yaml":"examples/fail_patterns.yaml",
"metrics_out":"out/metrics_ts.csv","date":"auto"}})
if "an issue tracker" in s or "bug" in s:
T.append({"task":"jira_fetch","params":{"open_jql":"...", "completed_jql":"..."}})
if any(k in s for k in ["forecast","p80","p50","finish","date"]):
T.append({"task":"forecast_dual","params":{"metrics_ts_csv":"out/metrics_ts.csv",
"code_target":0.95,"func_target":0.95,"slots":64}})
T.append({"task":"report","params":{}})
return {"tasks": T}
```

Listing 2. Planner fallback (intent → strict JSON plan)

The orchestrator reads the plan and executes tasks one by one via the registry. After each call it stores the agent's outputs in a shared context and forwards what later steps need—for example, open-bug count and average fix days are threaded into the forecaster call. Coverage merge/report and metrics append are idempotent, and the reporter always rewrites a concise status page so there's a consistent place to read "what happened." Side effects only occur when the execution gate is set; hard errors stop the run with a clear message, while non-critical lookups can be skipped so a status and forecast still appear from the data at hand.

```
# simple_orchestrator.py (excerpt)
from .planner import plan_from_intent
from .registry import Registry
def execute_plan(intent: str) -> dict:
plan = plan_from_intent(intent)
ctx = {"artifacts": {}}
for step in plan.get("tasks", []):
name = step["task"]; params = step.get("params", {})
# thread an issue tracker metrics into forecast params if present
if name == "forecast_dual":
params["bugs_open"]   = ctx.get("bugs_open", 0)
params["avg_fix_days"] = ctx.get("avg_fix_days", 0.0)
out = Registry[name](params)
ctx["artifacts"][name] = out
if name == "jira_fetch":
ctx["bugs_open"]   = out.get("open_bugs", 0)
ctx["avg_fix_days"] = out.get("avg_fix_days", 0.0)
```

```
if "report_md" in out:
ctx["report_md"] = out["report_md"]
return {"plan": plan, **ctx}
```

Listing 3. Orchestrator core (sequential execution, typed I/O)

In current Etabot, P50/P80 dates come from Monte Carlo simulation over a reset-aware forecast model. The forecaster consumes the metrics time series (Appendix A) and fits recent progress rates for code and functional coverage; if coverage drops due to certain changes (e.g., new coverpoints), it detects the step and refits on the post-reset segment. Sampling yields a completion-time distribution from which P50 (median) and P80 (conservative) finish dates are reported. The default model uses only the within-project trace, but historical or cross-project predictors can be adopted by swapping the forecaster behind the same typed interface.

V. MIDPOINT DATA AND USER EXPERIENCE

Midpoint visibility is central to Etabot. The CLI and the onprem chatbot expose a consistent set of status primitives, making the same information available whether a user types a command or asks a question in natural language:

Latest pass rate — the current designonly pass rate (infrastructure failures excluded), computed over the configured window.

Coverage status — the latest code and functional coverage from the most recent merged report.

Open bugs — current openissue count and average fix time from the issue tracker.

Forecast — summarized P50/P80 finish dates for the configured coverage targets.

In snapshot mode, queries are answered from the most recent onprem artifacts (metrics timeseries, coverage report, issuetracker stats). When autorun is enabled, naturallanguage questions can trigger the entire agent flow before replying. For example:

“What’s the current P80 date?” The system plans a small, typed workflow, then executes it end-to-end: submit regression jobs, optionally wait for them to complete, merge coverage, compute the designonly pass rate, query open bugs and average fix days, run the forecast, and finally render a concise report. The reply includes the updated P50/P80 dates and a short status summary; the same artifacts are published as onprem resources for tools and assistants to read. Through the same interface, teams can check midpoint health on demand or ask the system to refresh it by running the full pipeline from a single, human readable instruction.

```
Midpoint status (latest artifacts):
- design_only_pass_rate_7d: 0.93
- code_coverage:            0.80
- functional_coverage:      0.70
- open_bugs:                12
- avg_fix_time_days:        3.8
Forecast to code=0.95, func=0.95:
- code:     P50 2025-08-19 | P80 2025-08-23
- func:     P50 2025-09-05 | P80 2025-09-14
- overall:  P50 2025-09-05 | P80 2025-09-14
```

Listing 4. Example Etabot output

Impact can be quantified along two axes. First, status preparation is reduced from a multi-step manual workflow (regression launch, coverage merge/report, parsing, spreadsheet updates, bug queries, and manual write-up) to a single typed intent that deterministically produces the same on-prem artifacts (result.csv, metrics_ts.csv, report.md). Second, forecast quality can be evaluated by walk-forward backtesting: at each historical snapshot, predict a completion distribution and measure median absolute error of P50 and calibration, such as the fraction of runs where the actual closure date falls on or before P80.

VI. EXTENSIBILITY AND DATA SAFETY

Adapters isolate vendor specifics: simulators/schedulers in Runner agent; coverage backends in Coverage agent; bug tracker in Bug agent; forecasting model in Forecaster agent. Teams can switch simulators or schedulers, change coverage tooling, or swap the forecasting model while keeping planners, orchestrator, and MCP/chat interfaces unchanged. The Bug agent is behind a minimal "Bug API" (open-bug count and average fix time); the default adapter uses Jira/JQL, but other ticketing systems can be supported by swapping this adapter without changing orchestration or reporting.

The system is local-first: side-effectful commands are rendered by default (dry-run) and no data egress occurs. Administrators can enable execution explicitly. LLM planning is optional and can be restricted to on-prem servers. The MCP server may bind to localhost, keeping resources within the firewall.

VII. CONCLUSION

By focusing on orchestration and midpoint visibility, Etabot reduces status-prep time, preserves on-prem data residency, and maintains flexibility: agents are small, typed, and replaceable; the forecaster is a component, not a commitment. The work has limitations: Etabot does not attempt automatic coverage closure or stimulus optimization, and coverage interchange still depends on what backends (vendor tools or custom database readers) can faithfully represent in given environment. Future work includes adapters for other simulators and schedulers, database-native readers that avoid text parsing where tool support permits, and dashboards backed by the same MCP resources.

REFERENCES

[1] Anthropic, “Introducing the Model Context Protocol,” Nov. 25, 2024. [Online]. Available: https://www.anthropic.com/news/model-context-protocol.

[2] Model Context Protocol, “What is the Model Context Protocol (MCP)?,” [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro.

[3] E. Ohana, “Closing Functional Coverage With Deep Reinforcement Learning: A Compression Encoder Example,” DVCon Proceedings Archive, United States, 2023. [Online]. Available: https://dvcon-proceedings.org/document/closing-functional-coverage-with-deep-reinforcement-learning-a-compression-encoder-example/.

[4] J. Pluciński, Ł. Bielecki, R. Synoczek, E. Andersson, A. Löytynoja, and C. Macario, “Accelerate Functional Coverage Closure Using Machine-Learning-Based Test Selection,” DVCon Proceedings Archive, Europe, 2023. [Online]. Available: https://dvcon-proceedings.org/document/accelerate-functional-coverage-closure-using-machine-learning-based-test-selection/.

[5] Synopsys, “Accelerating Coverage Closure with AI-Based Verification Space Optimization,” White Paper. [Online]. Available: https://www.synopsys.com/verification/resources/whitepapers/vso-ai-wp.html.

[6] S. Kumari, D. N. Gadde, and A. Kumar, “Optimizing Coverage-Driven Verification Using Machine Learning and PyUVM: A Novel Approach,” arXiv:2503.11666, Feb. 2025. [Online]. Available: https://arxiv.org/abs/2503.11666.

[7] S. Siderova, “Monte Carlo Simulation Explained: Everything You Need to Know to Make Accurate Delivery Forecasts,” Nave, blog post. [Online]. Available: https://getnave.com/blog/monte-carlo-simulation-explained/.

APPENDIX A — EXAMPLE METRICS_TS.CSV

```
date,ccov_pct,fcov_pct,passrate

2025-08-01,0.40,0.70,0.91

2025-08-05,0.65,0.72,0.93

2025-08-10,0.72,0.66,0.91   # functional reset (new coverpoints)

2025-08-15,0.80,0.70,0.93
```

APPENDIX B — EXAMPLE RESULTS.CSV

```
date,test,status,fail_type,component,log

2025-08-15,uvm_smoke_001,PASS,NA,env,/runs/001/log.txt

2025-08-15,uvm_smoke_002,FAIL,infra,license,/runs/002/log.txt

2025-08-15,uvm_rand_013,FAIL,design,scoreboard,/runs/013/log.txt

2025-08-15,uvm_rand_014,PASS,NA,env,/runs/014/log.txt

2025-08-14,uvm_seq_031,FAIL,infra,timeout,/runs/031/log.txt

2025-08-14,uvm_seq_032,FAIL,design,assertion,/runs/032/log.txt

2025-08-13,uvm_reg_021,PASS,NA,env,/runs/021/log.txt

2025-08-13,uvm_reg_022,FAIL,design,uvm_error,/runs/022/log.txt
```