Paper #153 - IEA/AIE 2026

Emerging Trends and Challenges Toward LLM Agents in Static Analysis

Hoang-Quoc-Bao Hua • **Xuan-Bach Le**

Faculty of Computer Science and Engineering, HCMUT, VNU-HCM

Static Analysis

Analyzing source code without execution to guarantee soundness and early bug detection

1. What is it?

Static Application Security Testing (SAST) inspects program source code without executing it. It parses raw files into Abstract Syntax Trees (AST) and Control Flow Graphs (CFG) to mathematically reason about execution paths.

2. Why it is Powerful

It guarantees complete path coverage (soundness) to catch obscure edge cases dynamic testing misses. By "shifting left" in the lifecycle, it identifies vulnerabilities early when they are 10-100x cheaper to fix.

Static Analysis in Action

Example: Detecting Null Pointer Dereference

```
public String getProfile(User user) {
    // Code path where 'user' can be null
    if (user == null) {
        log.warn("Null user context");
    }
    ❌ return user.getName();
⚠️ SAST: 'user' is dereferenced without null check
}
```

Motivation: Why Static Analysis is Brittle

Though powerful, traditional SAST struggles with major practical limitations

1. **High false positives**

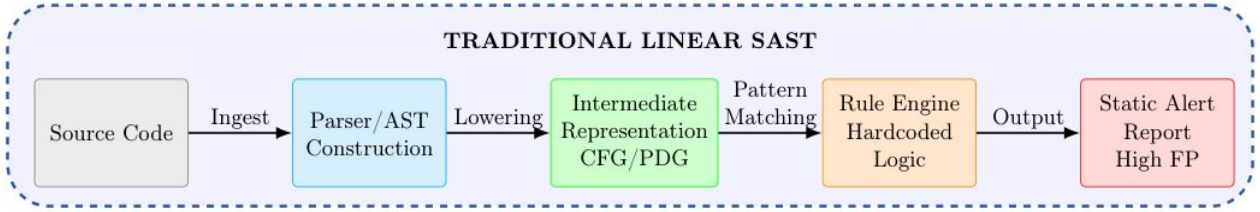
   Conservative over-approximation protects soundness, but creates alert fatigue and manual triage burden.

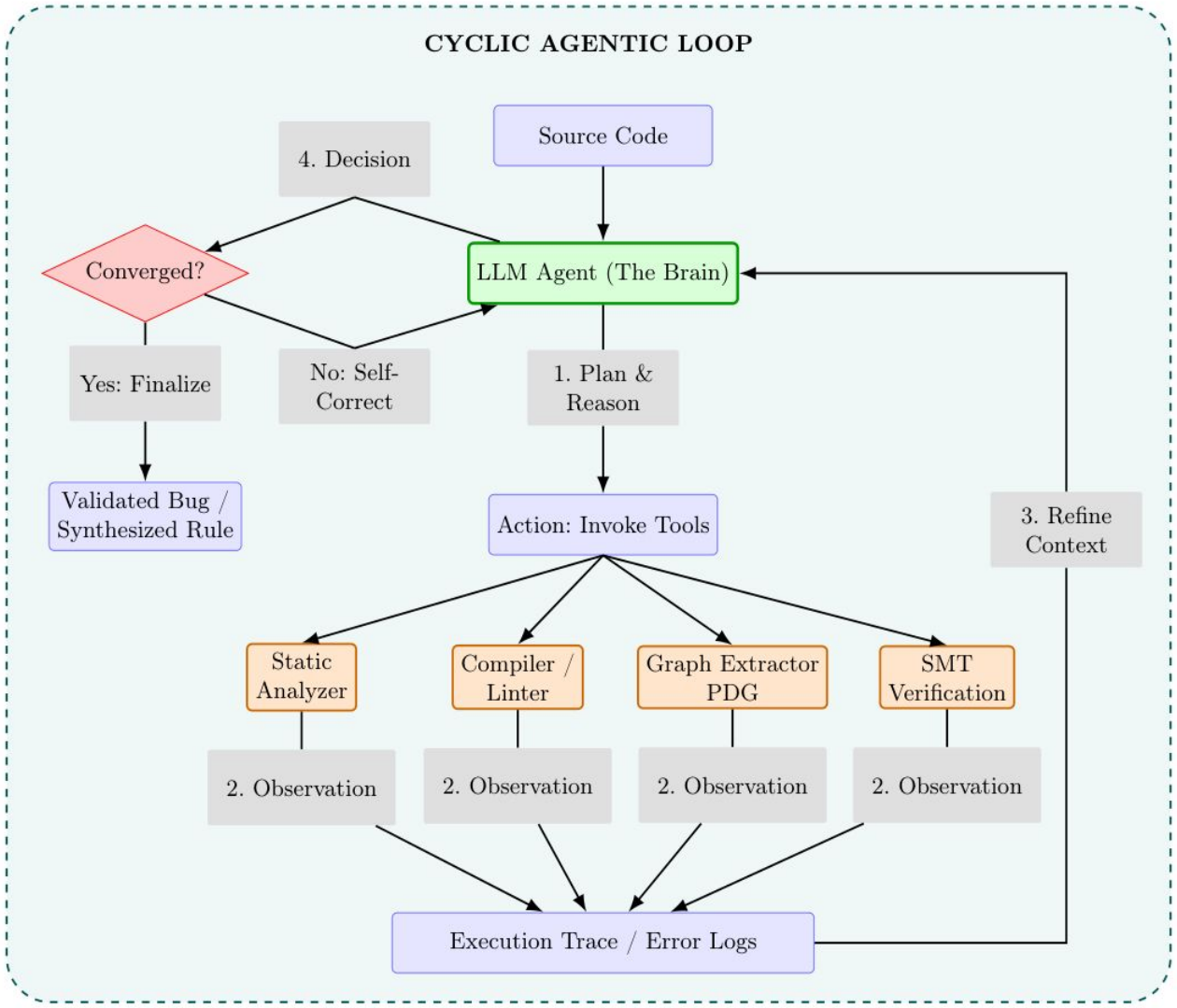
2. **Rigid rule sets**

   Traditional analyzers encode known patterns; they struggle with implicit intent, custom libraries, and evolving systems.

3. **Weak semantic context**

   Some warnings require hypothesis-driven reasoning across code, dependencies, runtime traces, and environment assumptions.





The Shift: Scanning to Reasoning

Moving from rigid linear pipelines to iterative agentic loops

Traditional SAST

A linear pipeline parses code, lowers it to intermediate representations, applies fixed rules, and emits alerts.

Agentic SA

An LLM agent plans, invokes tools, observes feedback, refines context, and decides whether the result has converged.

The Core Paradigm Shift

The shift is not "LLM instead of analyzer" — it is "LLM coordinating analyzers, compilers, graphs, and solvers."

Taxonomy

four dimensions of agentic static analysis

1. Architecture

Single-agent ReAct loops or multi-agent collaboration.

2. Integration pattern

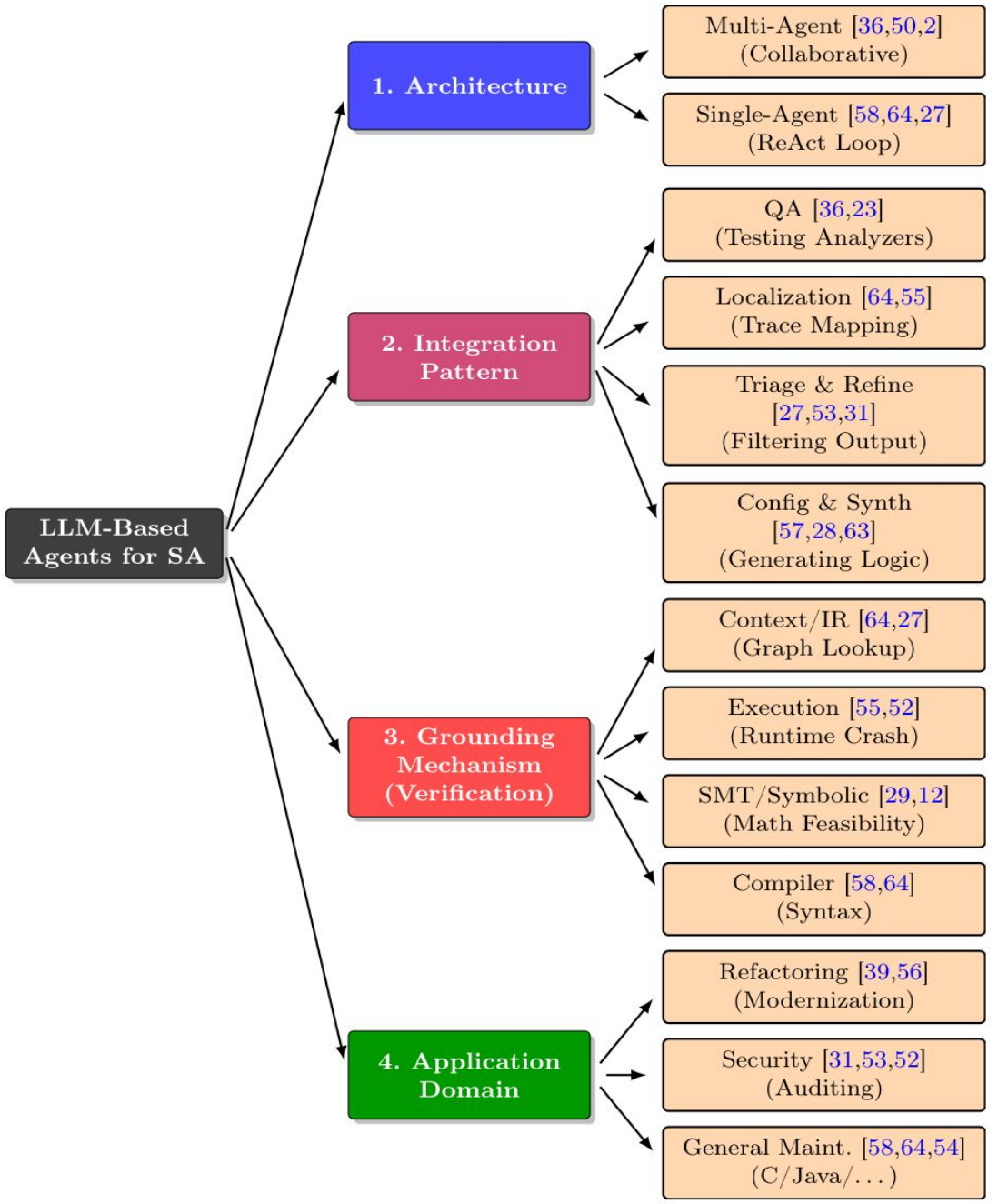
Where the agent intervenes: synthesis, triage, localization, or testing analyzers.

3. Grounding mechanism

How reasoning is validated: compiler, SMT, runtime traces, or static graphs.

4. Application domain

Security auditing, maintenance/refactoring, or static-analysis tool development.



Dimension 1 - Architecture

Single-agent Architectures

One LLM coordinates the loop: **plan act observe refine**.

Representative examples include **ZeroFalse**, **T2L-Agent**, and **AutoCodeRover**.

Multi-agent Architectures

Specialized agents split work across **generation, validation, testing, and evaluation**.

Examples include **StaAgent** and **MAGIS**.

Core Design Choice

Reasoning granularity, collaboration, and scalability.

Dimension 2 — Integration pattern

Where does the agent enter the analysis workflow?

01

Config & synthesis

Generate rules, checkers, or queries from natural language and historical patches.

02

Triage & refine

Filter static alerts using context, constraints, traces, or inferred specifications.

03

Localization

Map runtime failures, stack traces, and tests back to relevant static code regions.

04

QA / analysis

Test static analyzers by generating variants and detecting inconsistent behavior.

Dimension 3 — Grounding mechanism

Grounding: how agents avoid hallucinated analysis

01

Compiler

Checks syntax, typing, and build-level validity.

02

SMT / symbolic

Converts behavior into constraints for feasibility reasoning.

03

Execution

Runs tests, traces, or debuggers to validate behavior.

04

Context / IR

Uses call graphs, PDGs, specs, and repository context.

Dimension 4 — Application Domain

Primary domains where LLM-based static analysis agents are deployed

01

General maintenance

Target tasks such as bug fixing, fault localization, and code improvement.

02

Security auditing

Focus on vulnerability detection, false-positive reduction, and taint reasoning.

03

Refactoring & modernization

Transform code while preserving behavior and modernizing legacy systems.

04

Tool development

Generate checkers, rules, or tests for static analyzers themselves.

Representative systems through the taxonomy

System	Architecture	Integration	Grounding	Domain	Key Insight
Knighter	Single-agent	Logic synthesis	Compiler	Tool dev.	Learn checkers from patches
RuleLLM	Single-agent	Rule synthesis	Compiler	Security	Iterate rule repair
LLift	Single-agent	Triage	Constraints	Security	Feasibility-based FP reduction
ZeroFalse	Single-agent	Triage	Context + trace	Security	Contract-style warning verification
IRIS	Single-agent	Triage	Spec inference	Security	Library-aware taint reasoning
T2L-Agent	Hierarchical	Localization	Runtime trace	Security	Maps crashes to static code
AutoCodeRover	Single-agent	Localization	Execution	Maintenance	Test-guided navigation
ConSynergy	Hybrid	Verification	SMT	Security	SMT-based path reasoning
AutoBug	Hybrid	Symbolic execution	SMT	Maintenance	LLM-assisted symbolic execution
StaAgent	Multi-agent	Tool testing	Metamorphic	Tool dev.	Finds analyzer inconsistencies

Applications

What LLM agents are doing in practice

Rule synthesis

Translate vulnerability descriptions or historical patches into executable rules/checkers.

Autonomous localization

Use runtime signals and repository search to narrow analysis to relevant code regions.

False-positive adjudication

Treat warnings as hypotheses and try to prove feasibility or infeasibility.

Meta-analysis of analyzers

Generate semantically equivalent code variants to reveal analyzer inconsistencies.

Benchmarks: Accuracy is Not Enough

Agentic static analysis needs workflow-level evaluation.

Old Paradigm

Did it find the bug?

New Paradigm

Did it reason **reliably**, **cheaply**, and **audibly**?

Things to be benchmark:

1. **Reasoning quality**
2. **Tool behavior**
3. **Reliability**
4. **Cost**

Current benchmarks help — SWE-bench, Defects4J, LoCoBench, ReliabilityBench — but do not yet fully capture agentic static analysis (SA).

Research Gaps: Evaluation and deployment

Production needs controls around non-determinism, context limits, attacks, and auditability.

10–15×

Cost Increase

higher cost reported for agentic workflows versus traditional tools

15–25%

Variance

classification variance across runs in some agentic settings

Evaluation

Benchmark Gap

Function-level benchmarks miss repository-scale context, tool coordination, and assumption handling.

Deployment

Deployment Gap

Safety-critical settings need auditability, determinism, fallback mechanisms, and human review.

Takeaways: reliable agents need safeguards

01

Make assumptions explicit

Agents should output environmental and logical assumptions as checkable artifacts before generating code or final judgments.

02

Use multi-modal grounding

Cross-check static constraints with dynamic validation to expose operational blind spots.

03

Keep human in loop

When confidence drops, fall back to conservative analysis and escalate unresolved assumptions to human reviewers.

Got questions or want to discuss the findings?

Questions & Discussion