AI Agents in Software Engineering: From Hype to Production

Srinath Srinivasan

How many of you used an LLM to write code this week?

Probably most of you.

What you're used to:

- Asking Copilot to autocomplete a function
- Using ChatGPT to debug a cryptic error message
- Prompting Claude to explain a concept

These are **assistants**. You're still driving.

What's already in production:

- AI agents that *autonomously* triage bug reports
- Agents that write, run, and fix their own tests
- Agents that file bug reports to engine maintainers

These are **agents**. They drive themselves.

Today: what AI agents actually look like in a real SE research system, what they're good at, what they can't do, and what it means for you.

What Even Is an AI Agent?

An LLM by itself is a text transformer. An agent is a text transformer that can *act*.

LLM Alone

- Takes text in
- Produces text out
- No memory between calls
- No external access

$\Rightarrow$

LLM + Tools

- Can call a terminal
- Can query a database
- Can search the web
- Can read a spec document

$\Rightarrow$

Multi-Agent System

- Orchestrator delegates to sub-agents
- Each agent has a focused role
- Agents share findings
- System adapts to complexity

The Three Ingredients of an Agent

(1) A model that reasons **(2) Tools that give it access to ground truth** **(3) A loop that lets it act, observe, and revise**

Agents Are Entering SE Right Now

Code Generation

- SWE-bench: agents autonomously resolve GitHub issues [Jimenez24 ICLR]
- Devin, SWE-agent: full repo-level task completion

Testing & Triage

- Automated test generation from specs
- Agent-driven regression classification
- **Differential fuzzing triage** ← *today's case study*

Incident Management

- automated alarm triage at cloud scale
- Triage agents that cross-reference runbooks

Code Review

- PR review bots that check style, security, logic
- Agents that suggest refactors with test coverage

The Paradigm Shift: From Copilots to Infrastructure

Synchronous AI

- Lives in your IDE or browser
- You prompt it, you wait for the answer
- It assists *one human at a time*
- Examples: GitHub Copilot, ChatGPT

The limitation: It is bottlenecked by how fast you can type and read.

Asynchronous AI

- Lives in the CI/CD pipeline
- Wakes up on event triggers (e.g., a PR merge)
- Swarms the problem in the background
- Examples: Code review bots

The advantage: It scales compute infinitely while you sleep.

The future of SE isn't talking to AI more.
It's building infrastructure where AI talks to other systems in the background.

Before We Dive In: The Elephant in the Room

"Will agents take my job?"

The history of SE tooling:

- **Compilers** automated assembly writing
- **IDEs** automated syntax checking
- **CI/CD** automated deployment
- **Agents** are automating cognitive grunt work

Nobody thinks compilers replaced programmers. They changed *what programmers spend time on*.

The short answer:

- Not in the way you're imagining
- The *tedious* is being automated
- The *judgment* is staying human
- The *stakes* on human decisions are rising

We'll come back to this at the end with a concrete framework.

Software Testing at Scale: The Numbers

Before we look at the solution, feel the size of the problem.

What production testing looks like:

- A single fuzzing campaign: **50 CPU cores, 48 hours**
- Output: **10,000+ potential bug reports**
- Reality: **fewer than 100 are real bugs**
- That is a **100:1 noise ratio**

The human cost without automation:

- Expert review: **30+ minutes per finding**
- 10,000 findings $\times$ 30 min = **5,000 person-hours**
- At \$55/hr: **\$275,000 per campaign**
- And the spec changes next month

The Core Question

If you had 10,000 potential bugs to review and only 100 are real, what do you do?

You need an oracle — something that judges each finding and says: *real bug* or *noise*.

Differential Fuzzing: How It Works

Fuzzing [Miller90]

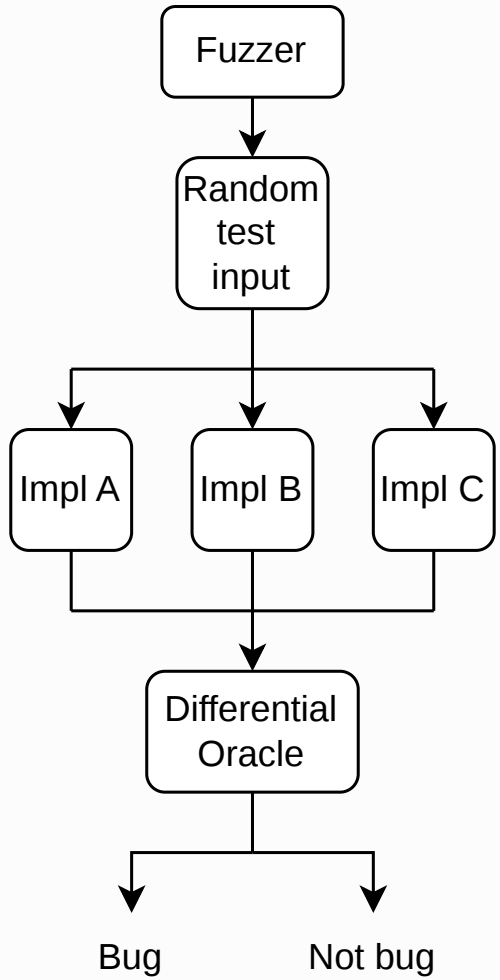
- Automatically generate massive volumes of random inputs
- Run them against a target program, look for crashes or misbehavior

Differential Testing [McKeeman98]

- Feed the *same* input to **multiple implementations** of the same spec
- Disagreement between implementations signals a potential functional bug

Differential Oracle

- The *judge*: decides whether a discrepancy is a **true bug**
- This is the hardest part [Dinella22 ICSE], and the bottleneck we target



Think of it like giving the same math problem to three calculators and checking if they all agree.

Motivation: Why Oracles Are the Bottleneck

Generating inputs is a solved problem. Judging them is not. [Barr14 TSE]

- **Overwhelming Volume:** 50 CPU cores for 2 days generates **10,000+ differential findings**.
- **High Noise:** Most findings are not bugs; they are allowed behaviors, undefined behaviors, or spec evolution [Lima21].
- **Expensive Triage:** Manual review takes **30+ minutes per finding** for domain experts [Wachter25 NDSS].
- **Brittle Automation:** Traditional rule-based filters break every time a specification changes or a new edge-case is discovered [Bushart23].

At modern fuzzing scales, the bottleneck is no longer finding bugs. It is filtering noise.

The Stress Test: JavaScript Engines

JavaScript engines are one of the most hostile differential testing environments in existence.

Inescapable Scale:

- Powers **98.7% of all websites** [W3Techs25]
- Four distinct production engines: V8 (Chrome), JSC (Safari), SpiderMonkey (Firefox), GraalJS
- Signal-to-noise ratio exceeding **100:1** in production fuzzing

Why it's so hard:

- Engines are **decades-mature**. Surface bugs were found long ago.
- Valid new bugs require hitting hyper-specific edge cases in the spec
- Engines execute code **trillions of times per day** — even a small bug has global reach

Why Rule-Based Oracles Collapse

In JavaScript engines, **divergence is often within spec**. Rule-based filters can't keep up with the number of valid variations. (e.g., JSC's `write()` hits the file system, V8's prints to console).

```
function test() { return 42; }
var bound = test.bind(null);

print("Regular function:");
print(test.toString());
print("\nBound function:");
print(bound.toString());
```



V8

```
function test() { return 42; }
function () { [native code] }
```


Spidermonkey

```
function test() { return 42; }
function() {
    [native code]
}
```



JavaScriptCore

```
function test() { return 42; }
function test() {
    [native code]
}
```

All three outputs are correct. None are bugs.

Rule-based oracles fail because they cannot adapt to moving targets.
Every time the spec evolves, someone has to manually update the rules.

Let's Look at a Real Divergence: The write Function

Why is noise so hard to filter? Let's look at a concrete example.

The Fuzzer Output:

```
write("Hello World");
```

V8 (Chrome):

```
Prints to standard out
```

JSC (Safari):

```
Attempts to write to a local file
system path and throws an error
```

Is this a bug?

- A rule-based fuzzer sees: **Different output → BUG!**
- **The Reality:** `write()` is a host-defined environment function, not part of strict ECMA-262.
- **The Verdict:** **Not a bug. This is allowed divergence.**

If your system flags this as a bug, your False Positive rate explodes.
Our AI Oracle needs to be smart enough to read the spec and recognize host-defined exceptions.

Why Not Just Ask ChatGPT?

This is the right instinct. Here is why it breaks at scale.

What a single LLM call can do:

- Read a bug report and make a judgment
- Explain why a behavior might be wrong
- Suggest a fix

...for one finding, in isolation.

Where it breaks down:

- **Hallucination** when juggling the spec, the code, and the history simultaneously
- **No execution**: it can't actually run the JavaScript to verify
- **No memory**: it can't check if it's seen this bug before
- **No spec access**: it's working from training data, not the actual ECMA-262

The Critical Failure Mode

A monolithic LLM will confidently tell you a finding is a real bug when it is not because it cannot *check*. It reasons from pattern-matching, not from ground truth.

The solution: give the model tools, and break the task into focused roles.

General-Purpose Agents: The Promise

Tools like Claude Code, OpenAI Codex, and Cursor raised a natural question:

"Why build a custom agent? Can't we just use one of these?"

What general-purpose agents promise:

- A single agent that can reason about any codebase (massive context)
- Pre-integrated tools: terminal, browser, file system
- Natural language task specification
- Zero infrastructure to build

What they're actually built for:

- Interactive, one-shot developer assistance
- Open-ended tasks with a human in the loop
- Scenarios where latency is seconds, not milliseconds
- Low-volume, high-variance work

These are powerful tools. But "powerful" and "right for this job" are not the same thing.

General-Purpose Agents at Scale: Where They Break

Three concrete failure modes when you run a general-purpose agent at production fuzzing scale.

Cost

Claude Code / Codex use frontier models by default.

GPT-4o: **~$0.04–$0.12 per task** depending on context length.

10,000 findings × $0.04 = **$400–$1,200 per campaign**.

SmartOracle on Gemini Flash: **$30 total**.

Latency

General-purpose agents load large tool contexts, browsing agents, full repo state.

Typical latency: **60–120 seconds per task**.

10,000 findings at 90s each = **250+ CPU-hours**.

SmartOracle median: **20 seconds**.

Black Box

Commercial agents don't expose reasoning traces.

You cannot audit *why* a finding was classified True or False.

For bug reports filed to engine maintainers: **no trace = no credibility**.

SmartOracle logs **every tool call and sub-agent decision**.

The Structural Mismatch

General-purpose agents are optimized for **interactive, low-volume, human-supervised** tasks.
High-throughput automated triage needs **batch-optimized, auditable, domain-specialized** systems.
These are different design targets, not just different parameter counts.

Industry Evidence: Cost and Auditability Drive Specialization

This isn't just a research concern. Enterprise AI adoption is hitting the same wall.

The Cost Cliff:

- Andreessen Horowitz (2024): AI-native companies spending **60–80% of revenue on inference** before optimization [a16z24]
- The response: specialized, quantized, fine-tuned models replacing frontier API calls for repetitive tasks
- **"The winning move is a small model that knows one domain deeply, not a large model that knows everything loosely"**

The Auditability Requirement:

- EU AI Act (2024): high-risk AI systems must maintain **audit logs of automated decisions** [EUAIAct24]
- Financial services, healthcare, safety-critical software: regulators are starting to ask *why* an AI made a decision
- A black-box general agent **cannot answer that question**
- SmartOracle's logged reasoning traces are an early version of this compliance posture

Specialization is not an academic choice. It is how industry survives the cost cliff
and satisfies the coming auditability requirements.

Decompose, Don't Monolith

The Agentic Insight:

- Fuzzing triage requires distinct cognitive phases: *isolate, check spec, verify, challenge, deduplicate*
- We reverse-engineered human expert debugging behaviors [Layman13]
- Strict "role-playing" across focused sub-agents drastically reduces hallucination [Gosmar25]

The Human Analogy

A senior engineer debugging a critical production issue does not do everything at once. They isolate the symptom, check the spec, challenge their own conclusion, then check for duplicates. We encode these phases explicitly.

Why Prompt Chaining Fails:

- A rigid sequential prompt forces a pre-scripted path
- Simple findings still pay the full reasoning cost
- Loses coherence when multiple constraints must be juggled simultaneously [Xu25]

What Agentic Orchestration Gives You:

- **Conditional execution**: skip steps for obvious cases
- **Dynamic tool use**: fetch spec only when needed
- **Focused context**: each agent sees only what it needs

SmartOracle emulates human expert workflow using specialized sub-agents

Class Exercise: Design the Oracle

Let's do some System Design.

The Constraints

- You have a single LLM.
- You have 10,000 bug reports to evaluate.
- You know that asking the LLM to do it all in one prompt causes hallucinations.

Discussion Questions:

1. If you could give this LLM *tools* (APIs it can call), what tools would it need?
2. How would you break the prompt down into a pipeline of smaller, specialized agents?
3. How do you prevent the AI from confidently lying?

(Class Discussion)

SmartOracle Architecture: Agents & Tools

Orchestrator:

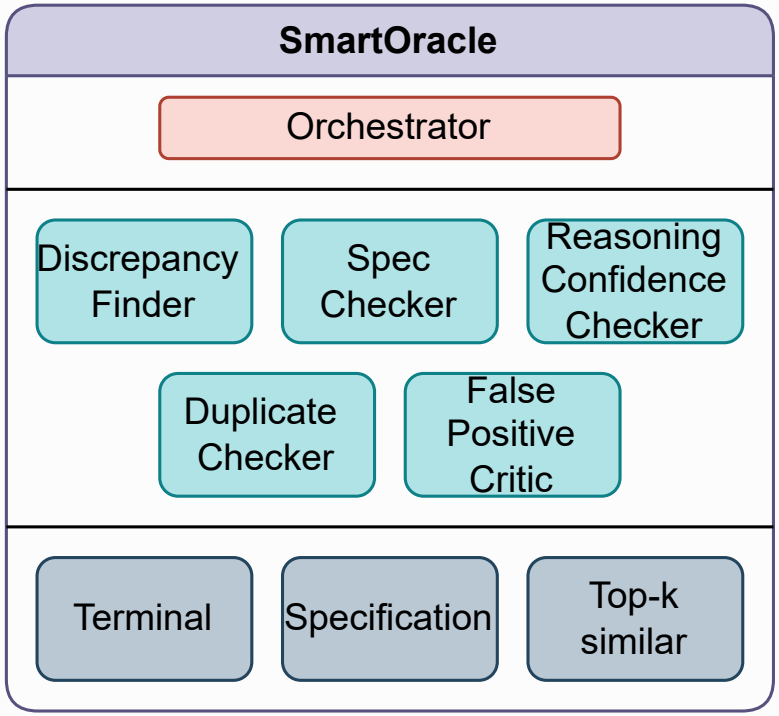
- Coordinates the workflow

5 Role-Playing Sub-Agents:

- **Discrepancy Finder**: Isolates the relevant code
- **Specification Checker**: Validates against ECMA-262
- **False Positive Critic**: Challenges the finding
- **Duplicate Checker**: Avoids re-reporting known bugs
- **Confidence Checker**: Quantifies evidence strength

3 Domain Tools:

- **Terminal**: Actually runs the JavaScript
- **Spec**: Queries cached ECMA-262
- **History**: Retrieves top-K past findings



Think of each agent as a job title

Discrepancy Finder = junior dev who isolates the symptom
Spec Checker = the person who actually reads the manual
FP Critic = the skeptical senior who asks “are you sure?”
Duplicate Checker = the PM who checks the issue tracker
Confidence Checker = the tech lead who signs off

Why the “Spec Checker” Agent is Necessary

You cannot use a Regex to understand ECMA-262.

ECMA-262 Section 11.9.1 (ASI):

> “When, as the source text is parsed from left to right, a token is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true: The offending token is separated from the previous token by at least one LineTerminator...”

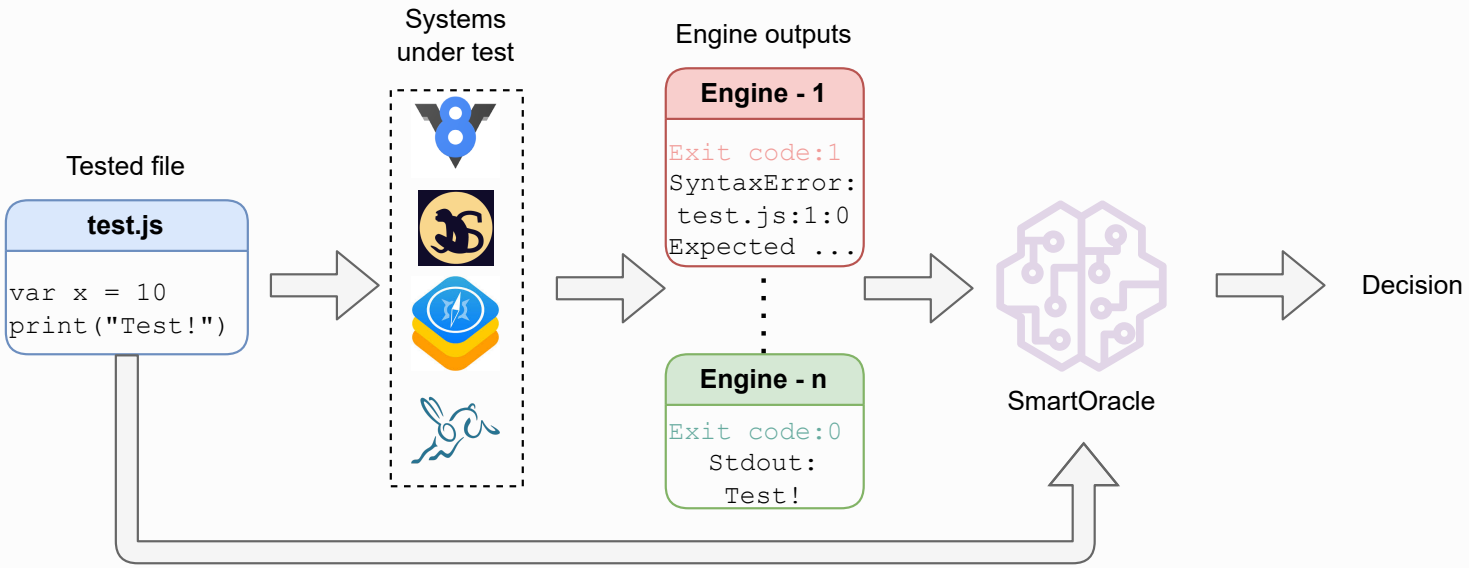
Why Rule-Based Logic Fails:

- The spec is written in dense, legalistic English, not math.
- It constantly references other nested sections.
- The rules evolve and change every year.

Why LLM Agents Succeed Here:

- LLMs excel at translating natural language constraints into execution logic. We let the agent read the English so we don’t have to code the rules.

The SmartOracle Workflow



Tools Over Agents: The Deeper Principle

This is the most important architectural lesson from this research.

What we assumed going in:

- Sub-agent coordination would drive quality
- The orchestration logic would be the key contribution
- More agents = better coverage of the problem

What the traces actually showed:

- **Tool invocations are 3× more frequent** than inter-agent calls
- Agents spend most time in **evidence-gathering cycles**, not talking to each other
- Removing an agent hurts less than removing a tool

The Generalizable Principle

In any agentic system you build, the bottleneck is usually not reasoning quality. It is **access to ground truth**.

An agent that can run code, query a database, or read a live document is fundamentally more reliable than one that reasons from memory alone.

Google DeepMind Validation [Tomašev26]

Feb 2026: current multi-agent delegation *"relies on simple heuristics"*. They are glorified task runners.

If you had to cut one thing: cut agents, keep tools.

How Do You Test a Bug-Finding System?

This is where research gets genuinely hard.

What you'd normally do:

- Split your labeled dataset into train/test
- Measure accuracy on the held-out set
- Report precision, recall, F1

The problem here:

- JavaScript bugs are **extremely rare**
- Nobody publishes a list of "non-bugs"
- Historical bugs can't be reproduced without version-pinned engine builds from years ago

The meta-problem:

You cannot evaluate a bug-finding oracle without first solving the labeling problem. Without ground truth, you cannot measure anything.

Our Solution

We built our own benchmark from a **live 48-hour fuzzing campaign**, then used a **semi-supervised pipeline** to generate ground truth at scale — labeling clusters, not individual findings.

Generating Ground Truth in a Label-Starved Domain

The Challenge: JavaScript is a label-starved domain. Bugs are hard to find, making evaluation impossible without manufacturing surrogate labels.

The Insight: We don't label everything. We label clusters.

The Semi-Supervised Pipeline:

1. **Generate:** 48-hour KITTEN [Xie25 ISSTA] campaign across 4 engines $\rightarrow$ **~10,000 findings**
2. **Group:** Segment findings by their **cross-engine exit code tuples**
3. **Cluster:** Apply **K-means** to group similar behavioral states within each tuple
4. **Propagate:** Human domain experts triage only the **cluster medoids**, propagating the label to the rest of the cluster

A massive labeled dataset at a fraction of the manual cost.

How K-Means Saved 5,000 Hours

Exit codes only tell part of the story. We needed semantic clustering.

The Problem:

- 500 different inputs all cause JSC to throw a `TypeError`.
- Are they the *same* bug?
- Some are `TypeError: undefined is not an object`.
- Others are `TypeError: readonly property`.

The ML Solution:

1. Convert the error messages into vector embeddings (TF-IDF or dense vectors).
2. Apply **K-Means Clustering** to group them in multi-dimensional space.
3. Find the **Medoid** (the exact center point of the cluster).
4. A human expert labels ONLY the medoid. We propagate the label to the rest.

Instead of reading 500 error messages, the expert reads **one**.
This is how you generate ground truth at scale in a label-starved domain.

RQ1: Can Semi-Supervised Labeling Replace Manual Triage?

1. The Binary Win (Bug vs. No-Bug)

- Grouping by exit codes alone achieves **98.7% accuracy**.
- *Takeaway:* Exit codes are a near-perfect first filter to separate signal from noise.

2. The Root Cause Problem (Why K-means is necessary)

- Exit codes are blunt: 1 exit code $\neq$ 1 root cause.
- Baseline propagation (exit codes alone) only achieves **77.3% accuracy** for identifying the actual failure mode.
- **K-means text clustering improves this to 89.1% (+11.8pp).**

Where K-means saves the pipeline:

Exit Code Scenario	Baseline	K-means	Gain
High Diversity (*1 pattern masking 9 distinct root causes*)	28.6%	**92.9%**	**+64.3pp**
Already Homogeneous (*1 pattern = 1 root cause*)	98.9%	99.3%	+0.4pp

K-means untangles messy clusters without over-segmenting clean ones.

RQ2: Can SmartOracle Find True Bugs?

0.84 Recall | **18% False Positive Rate**

Dataset	Engines in dataset	Bugs	Recall
JEST [Park21 ICSE]	V8, GraalJS, QJS, XS	44	**0.84**
Lima et al. [Lima21]	JSC, V8, ChakraCore, SM, Hermes	16	0.75
Manual (ours)	V8, SM, JSC, GraalJS	238	0.73

The FPR Hurdle: Evaluating Against Known Noise

Prior work evaluates oracles solely on *positive bugs* because non-bugs are never publicized. To rigorously measure noise suppression, we evaluated SmartOracle against a purpose-built negative set of **136 manually labeled, non-reportable findings**.

Result: SmartOracle correctly suppressed 112 of these findings, yielding an **18% FPR**.

RQ3: Real Bugs in Production Engines

SmartOracle found 8 previously unknown bugs across 3 engines:

Engine	Bug Summary	Status
GraalJS #931	Parsing: Incorrect SyntaxError due to ASI	**Fixed**
V8 #438787152	Object.prototype.__proto__ assignment	Confirmed
V8 #446261067	d8: new print() doesn't throw TypeError	Confirmed
V8 #446661133	d8: write() lacks input validation	Confirmed
JSC #297423	String.prototype.search TypeError	Pending triage
JSC #298112	__proto__ identifier TypeError	Pending triage
JSC #299296	jsc write() file system side effects	Pending triage
JSC #299441	__proto__ global object TypeError	Pending triage

Production Impact: GraalJS #931

SmartOracle discovered a `SyntaxError` divergence that maintainers **confirmed and patched in production**. This is a bug in the engine that runs JavaScript on real software being used by real people. SmartOracle found it autonomously.

A note on developer skepticism

Engine maintainers are increasingly wary of AI-assisted reports. Our JSC findings are **pending triage, not rejected**. This is an honest reflection of where the field is right now.

Closing the Loop: Auto-Remediation

SmartOracle finds and verifies the bug. What is the next logical step?

The Current Triage Pipeline:

1. Fuzzer generates crash
2. SmartOracle verifies it's a real bug
3. **SmartOracle files a GitHub Issue**
4. Human reads issue
5. Human writes patch
6. Human opens Pull Request

The Auto-Remediation Target:

1. Fuzzer generates crash
2. SmartOracle verifies it's a real bug
3. **Fixer Agent drafts a patch**
4. **Testing Agent runs test suite**
5. **Agent opens the Pull Request**
6. Human reviews and clicks Merge

The Trust Threshold

We can only allow agents to open Pull Requests if their False Positive Rate is incredibly low. If an agent spams maintainers with fake fixes, it gets banned. **This is why SmartOracle's 18% FPR is so critical**.

RQ4: Architecture Beats Scale

Head-to-head evaluation on the JEST dataset (44 bugs)

Method	Recall	Time	Cost
SmartOracle (Gemini Flash, agentic)	**0.84**	**20s**	**$0.003**
LRM + Prompt Chaining (Gemini Pro, monolithic)	0.68	91s	$0.040

+16pp recall. 4× faster. 10× cheaper.

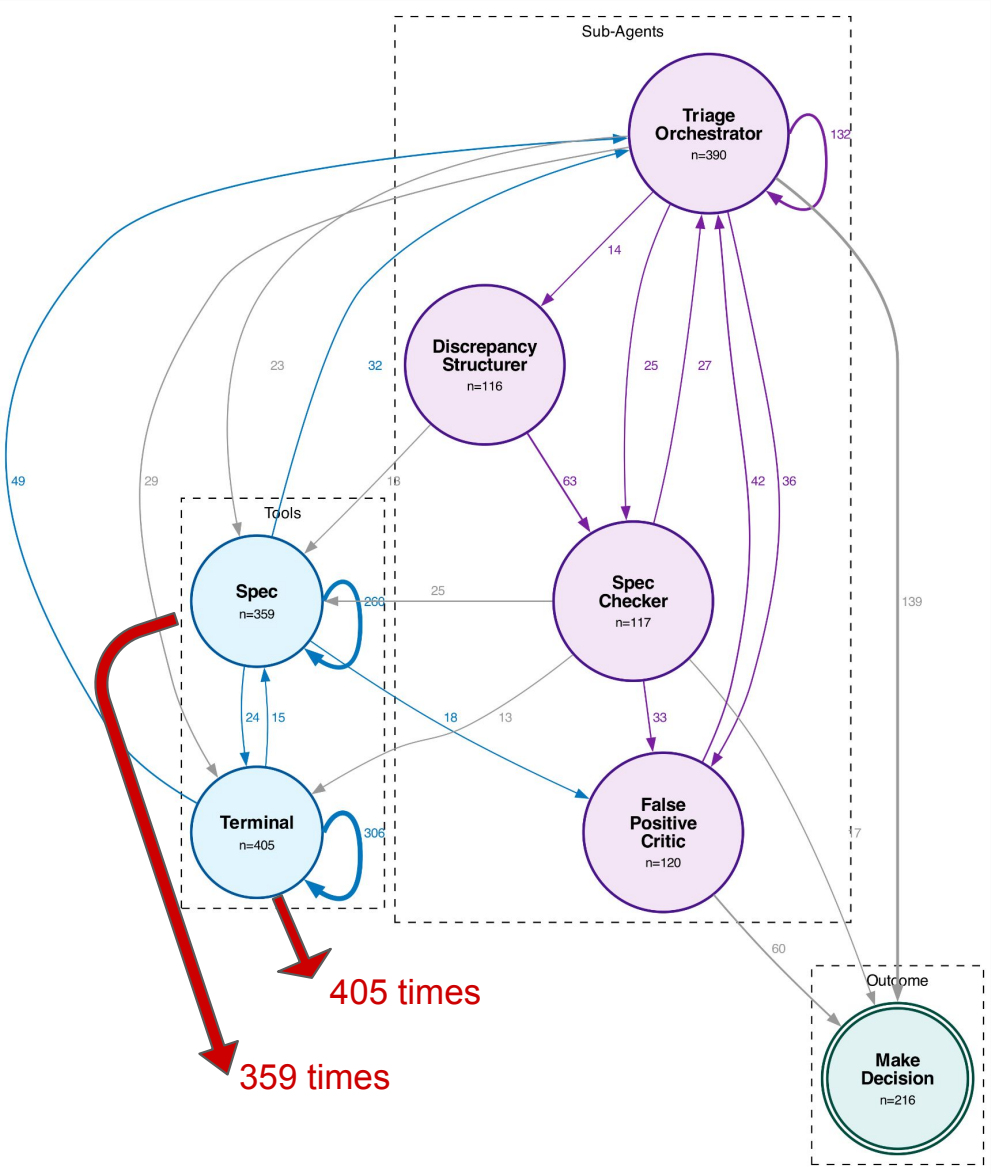
A smaller model with better architecture beats a massive model with rigid prompting.

Triaging our entire 10,000-finding campaign costs $30.

A domain expert at 30 min/finding would take **5,000 person-hours** costing **more than $250K**
SmartOracle does it overnight for the price of a dinner.

What the Traces Revealed: Tools Over Agents

216 logged reasoning traces. This is empirical, not a design assumption.



The thickness of transition lines tells the story: agents spend most time in **evidence-gathering cycles with tools**, not talking to each other.

Key insight: Rich tool interfaces matter more than adding more agents.

Decision Efficiency

Median: 6 steps — simple cases resolved fast.
95th pct: 18 steps — ambiguous cases trigger deeper deliberation.
SmartOracle scales compute to complexity. Monolithic models pay maximum cost for every query.

Conditional execution is the hidden cost lever.

Under the Hood: What is an LLM Tool?

We talk about "giving the AI a terminal." Here is what that looks like in code.

The LLM Schema Definition:

```
{
"name": "execute_javascript",
"description": "Runs JS code in V8...",
"parameters": {
"type": "object",
"properties": {
"code": { "type": "string" },
"engine": { "type": "string" }
},
"required": ["code", "engine"]
}
}
```

The Execution Loop:

1. The LLM outputs a **JSON string** instead of human text, requesting the tool.
2. The Orchestrator intercepts the JSON, halts the LLM, and runs the actual `node script.js` on the server.
3. The Orchestrator pastes the **stdout/stderr** back into the LLM's context window.
4. The LLM resumes reasoning.

The LLM never "touches" the terminal. It just knows how to ask for it.

MCP: The Universal Plug for Agent Tools

The tool schema we just saw is powerful — but every team was reinventing it differently. In 2024, Anthropic published a standard: ***Model Context Protocol (MCP)****.*

The Problem MCP Solves:

- Every agent framework (LangChain, AutoGen, CrewAI) defined tool interfaces differently
- Moving a tool from one framework to another required a full rewrite
- No standard for what a "server" exposes to an agent

What MCP Defines:

- A **client–server protocol**: the agent is the client, the tool is the server
- Servers expose: **Tools** (actions), **Resources** (data), **Prompts** (reusable instructions)
- Transport: stdio or HTTP/SSE — works locally or over a network

MCP in One Sentence

An MCP server is a process that speaks a standardized JSON-RPC dialect. Any MCP-compatible agent can discover what the server can do, call its tools, and get results — without knowing anything about the server's implementation.

Analogy:

- Before USB: every peripheral needed its own proprietary port
- After USB: one standard, any device
- **MCP is USB for agent tools**

As of 2025, MCP servers exist for: file systems, databases, GitHub, Slack, web browsers, Kubernetes, and more. [Anthropic25 MCP]

MCP in Practice: Integrating Tools the Standard Way

Here is what SmartOracle's Terminal tool would look like expressed as an MCP server.

MCP Server Definition (simplified):

```
# server.py (MCP terminal server)
@server.list_tools()
async def list_tools():
  return [Tool(
    name="execute_javascript",
    description="Run JS in a sandbox",
    inputSchema={
      "type": "object",
      "properties": {
        "code": {"type":"string"},
        "engine": {"type":"string"}
      }
    }
  )]
@server.call_tool()
async def call_tool(name, args):
  return run_in_sandbox(args["code"], args["engine"])
```

What the agent sees:

1. Agent queries the server: `list_tools()`
2. Server announces `execute_javascript` with its schema
3. Agent calls `call_tool("execute_javascript", {...})`
4. Server runs the code, returns stdout/stderr
5. Agent resumes reasoning with the result

The Key Benefit

The agent never changes. You can swap the JavaScript execution backend — Docker container, cloud VM, different engine version — without touching a single line of agent code.

Safeguards: Why Tools Must Run in a Sandbox

"Giving an AI a terminal" sounds powerful. It is also a significant attack surface.

What can go wrong without a sandbox:

- A hallucinating agent runs `rm -rf` on the wrong path
- Fuzzed JavaScript input contains a **prompt injection**: code that instructs the agent to exfiltrate data
- An engine exploit escalates from the tested JS process to the host OS
- A runaway loop burns through API quota or cloud compute

SmartOracle's safeguard stack:

- **Process isolation**: each JS execution runs in a separate child process with a timeout
- **Network block**: the execution environment has no outbound network access
- **Filesystem restriction**: the sandbox can only write to a designated temp directory
- **Resource limits**: `ulimit` caps CPU time and memory per invocation
- **Tool whitelisting**: the orchestrator only exposes the three defined tools — no shell expansion

The Design Principle: Minimal Capability

An agent should have access to exactly what it needs to do its job, and nothing else. Give it a terminal only if the terminal is itself sandboxed. Give it file access only to the directories it actually needs. This is the **principle of least privilege** applied to AI agents.

How Agents Show Their Work: Traces as Accountability

Sandboxing controls what the agent can do. Tracing controls what you can verify it did.

A logged SmartOracle trace (condensed):

```
[orchestrator] -> discrepancy_finder
[discrepancy_finder] -> tool: execute_javascript
  engine=V8, code={...}
  result: TypeError: x is not a function
[discrepancy_finder] -> tool: execute_javascript
  engine=JSC, code={...}
  result: (no output, exit 0)
[discrepancy_finder] -> spec_checker
[spec_checker] -> tool: query_spec
  query="TypeError callable check 7.3.12"
  result: {section text...}
[spec_checker] -> fp_critic
[fp_critic] decision: TRUE_BUG, conf=0.87
[confidence_checker] FINAL: TRUE_BUG
```

Why this matters:

- Every decision is **reproducible**: you can re-run the same trace
- Engine maintainers can see *exactly* what spec section was consulted
- Mistakes are debuggable: was it a bad tool result or a bad reasoning step?
- This is the **audit trail** that regulators and maintainers will increasingly require

From Research to Industry

216 traces logged in SmartOracle's evaluation. This is a prototype of the compliance infrastructure that enterprise agentic systems will need to build as standard practice.

Three Principles for Building with AI Agents

These aren't specific to fuzzing. Take these into every system you build.

1. **Decompose, don't monolith**
   Smaller, focused models with clear roles beat one massive model juggling everything. Hallucination drops dramatically when each agent has a single well-defined job.

2. **Tools are the moat**
   The value of your agent is determined by what it can *access*, not just what it can *reason*. A model that can run code, read a live spec, or query a database is categorically more reliable than one working from memory.

3. **Architecture beats scale**
   Good design with a smaller model outperforms brute-forcing with a bigger one. SmartOracle with Gemini Flash beat Gemini Pro at 10× lower cost. Don't reach for a larger model — reach for a smarter architecture.

These principles hold whether you're building a bug triage system, a customer support bot, a code review pipeline, or anything else involving an LLM with action capabilities.

The Automation Layer Cake: What Agents Actually Change

Not all engineering work is equally automatable. Here's a useful mental model.

Tier 1
Being Automated Now

High-volume triage
Boilerplate generation
Regression testing
Log analysis
Basic PR review
Test case synthesis

SmartOracle lives here.

Tier 2
Being Augmented (3–5 yrs)

API and system design
Security review
Cross-team coordination
Performance profiling
Dependency decisions

Agents suggest.
Humans decide.

Tier 3
Stubbornly Human

What does the user actually need?
Is this privacy tradeoff acceptable?
Mentoring a junior engineer
Navigating org politics
Accountability for failures

Requires social intelligence.

The profession is not disappearing. It is stratifying.

Engineers who thrive will *automate* Tier 1, *direct agents* in Tier 2, and *own* Tier 3.

From Fuzzing to the Cloud: The Enterprise Parallel

Why does industry care about agents?

JavaScript Fuzzing (Academia)

- 10,000 engine anomalies
- 100:1 noise ratio
- Ground truth: ECMA-262 Spec
- Action: File a bug report

Cloud Infrastructure:

- 10,000 automated server alarms
- 100:1 noise ratio
- Ground truth: Internal Runbooks
- Action: Page an on-call engineer

The operational problems are identical.

If you can build an agent system that triages JS engine bugs by autonomously reading a spec, you can build a system that triages datacenter outages by reading a runbook.

Paradigm shift, Part 1: Writing the Physics

We are moving from Imperative programming to Declarative orchestration.

The Imperative Era (Past/Present)

- You tell the computer **how** to do it.
- You write the loops, the logic, the specific error handling.
- You act as the translator between business logic and machine code.

The Declarative Era (Future)

- You tell the agent **what** the rules are.
- The agent figures out how to write the code to satisfy those rules.
- You act as the architect defining the boundaries of the system.

The most valuable skill won't be writing boilerplate code. It will be writing unambiguous, mathematically sound Specifications/designs.

If the spec is vague, the multi-agent system collapses. Humans define the rules; agents execute them.

Paradigm shift, Part 2: “Evals” Are the New Unit Tests

How do you test an employee that writes code slightly differently every time?

Traditional Unit Tests

- `assert(add(2, 2) == 4)`
- Deterministic: it passes or fails
- Tests strict, hardcoded logic paths
- Breaks easily when features change

Agent Evals (Evaluations)

- Behavioral benchmarks
- “Did the agent correctly parse the spec and avoid deleting production data?”
- Measures statistical reliability (e.g., Recall, False Positive Rate)

The Engineering Shift

In an agent-driven industry, human engineers will spend less time writing standard unit tests and far more time building **Evals**. Evaluating SmartOracle using K-Means clustering wasn’t just research—it is exactly how you will have to test autonomous systems in production. You don’t test the code; you test the agent’s behavior.

What This Means for You Specifically

Invest in these skills:

- **Directing agents, not just using them**
  Decomposing a complex task into agent-executable subtasks is a real, learnable skill. SmartOracle is a worked example.
- **Domain depth**
  Agents are generalists. The engineer who deeply understands the JavaScript spec is irreplaceable. Depth compounds.
- **Evaluation and evals**
  Knowing whether an AI agent is doing a good job is increasingly a core SE skill. Someone has to build the benchmark.

The honest uncertainty:

- Nobody knows exactly what the SE job market looks like in 10 years
- **Robust investments** across scenarios: systems thinking, communication, domain depth, critical evaluation of AI outputs
- SmartOracle's **18% FPR** is a concrete reminder — agents still make mistakes. Someone has to be accountable for catching them.

The Specification is Still Human

The spec tool is called thousands of times. But the spec itself was written by humans who reasoned carefully about what JavaScript *should* do. That upstream work — defining correct behavior — is not automatable.

Where Agentic SE Is Headed

Near-term (1–2 years):

- Agents that autonomously triage and route production incidents
- PR review bots with spec-level reasoning
- Automated test generation from natural language requirements
- Fuzz campaign orchestration with adaptive feedback

What won't change:

- The need for engineers who understand what *correct* means
- Accountability for shipped software
- The spec still needs to be written by humans

Medium-term (3–5 years):

- Human-in-the-loop systems: agent surfaces candidates, human approves
- Agents that maintain and update their own test suites as specs evolve
- Domain-specialized agents trained on proprietary codebases and internal docs

Discussion

What parts of a software engineer's job do you think are most automatable?
What requires the most human judgment, and why?

A2A: When Agents Need to Talk to Each Other

MCP solved how agents talk to tools. But what about how agents talk to other agents?

The new problem:

- SmartOracle's orchestrator calls its own sub-agents directly (same codebase, same memory space)
- But what if the Spec Checker agent is maintained by a different team?
- Or lives in a different cloud region?
- Or is a third-party service you're subscribing to?

Google's Agent-to-Agent (A2A) Protocol [Google25 A2A]:

- Open protocol released April 2025
- Defines how agents **discover, delegate, and return results** across organizational boundaries
- Built on HTTP + JSON, compatible with existing web infrastructure

A2A Core Concepts

Agent Card: A JSON manifest at `/.well-known/agent.json` that declares what an agent can do, its input/output schema, and its authentication requirements. Think: a résumé an agent publishes about itself.

Task: The unit of work. An orchestrator creates a Task, assigns it to a remote agent, and polls for status. Tasks can be long-running.

Artifact: The output an agent returns — structured data, files, or further agent references.

Co-developed by: Google, Atlassian, SAP, Salesforce, ServiceNow, and others. Designed to complement MCP, not replace it.

The Enterprise Swarm: A 3:00 AM Production Incident

Here is what the A2A protocol actually looks like in a modern enterprise.

1. **[3:00 AM]** Datadog detects a latency spike. It triggers the **Triage Agent**.
2. **[3:01 AM]** The Triage Agent checks recent logs. It suspects a database lock. Using A2A, it delegates a task to the **Database Agent**.
3. **[3:02 AM]** The DB Agent confirms a table lock and identifies the offending query. The Triage Agent then pings the **Git History Agent**.
4. **[3:03 AM]** The Git Agent finds that PR #4042, merged 10 minutes ago, introduced the query.
5. **[3:04 AM]** The Triage Agent pages the on-call Human.

PagerDuty Message to Human:
"Sev-1: Latency spike due to table lock. DB Agent isolated Query X. Git Agent mapped it to PR #4042 by [User]. **Press 1** to auto-revert the PR and kill the query. **Press 2** to open a secure terminal."

The agents did the 45 minutes of panic-driven forensics.
The human made the high-stakes business decision.

The Agentverse: A Vision of Composable AI Systems

MCP + A2A together sketch a picture of where the industry is heading.

Today

Agents are monolithic silos

Tools are hardcoded

Sub-agents live in the same codebase

Each company builds from scratch

No discoverability across teams

With MCP + A2A

Tools are plug-and-play servers

Agents publish their capabilities publicly

An orchestrator can *discover* a specialist agent at runtime

Cross-org delegation with authentication

Results are structured, auditable artifacts

The Vision

A “marketplace” of specialized agents

Your orchestrator routes sub-tasks to the best available specialist

Just as microservices replaced monolithic apps — **micro-agents replace monolithic prompts**

An important skill:
not writing agents, but *composing* them.

Understanding what agents can and can’t do, how to connect them safely, and how to evaluate whether the composition is trustworthy.

Standards Landscape

These protocols are maturing now. You will encounter them in industry within 2–3 years.

Protocol	Solves	How	Status
MCP Anthropic	Agent ↔ Tool communication	Standardized JSON-RPC server/client; servers expose Tools, Resources, Prompts	Open spec; widely adopted 2025
A2A Google et al.	Agent ↔ Agent delegation	HTTP task protocol; agents publish Agent Cards for discoverability	Open spec; v1 released Apr 2025
OpenAI Responses API OpenAI	Stateful, multi-turn agent runs	Persistent run state; tool results tracked server-side	GA 2025
LangGraph / LangSmith LangChain	Observability & tracing	Structured trace logging, evaluation, and replay for agent decisions	Open source

These protocols are converging on the same architectural pattern.
Specialized roles, tool-driven ground truth, auditable traces. The field is standardizing around what works.

References

- **[Barr14 TSE]** Barr et al., *IEEE TSE* 2014
- **[Miller90]** Miller et al., *CACM* 1990
- **[McKeeman98]** McKeeman, *Digital Technical J.* 1998
- **[Lima21]** Lima et al., *SQJ* 2021
- **[Markus24 ASE]** Fleischmann et al., *ASE* 2024
- **[Bernhard22 CCS]** Bernhard et al., *CCS* 2022
- **[Wachter25 NDSS]** Wachter et al., *NDSS* 2025
- **[Park21 ICSE]** Park et al., *ICSE* 2021
- **[Dinella22 ICSE]** Dinella et al., *ICSE* 2022
- **[Li25]** Li et al., *C&S* 2025
- **[Belcak25]** Belcak et al., *arXiv* 2025
- **[Tomašev26]** Tomašev et al., *arXiv:2602.11865* 2026
- **[Xu25]** Xu et al., *arXiv* 2025
- **[Tu21 ASE]** Tu & Menzies, *ASE* 2021
- **[Tu22 EMSE]** Tu & Menzies, *EMSE* 2022
- **[a16z24]** Andreessen Horowitz, *Emerging Architectures for LLM Applications* 2024
- **[EUAIAct24]** European Union, *EU Artificial Intelligence Act* 2024
- **[Jimenez24 ICLR]** Jimenez et al., *ICLR* 2024
- **[Yang24 NeurIPS]** Yang et al., *NeurIPS* 2024
- **[Hayet24 TSE]** Hayet et al., *IEEE TSE* 2024
- **[Xie25 ISSTA]** Xie et al., *ISSTA* 2025
- **[Layman13]** Layman et al., *EMSE* 2013
- **[Gosmar25]** Gosmar et al., *arXiv* 2025
- **[Anthropic25 MCP]** Anthropic, *Model Context Protocol Specification* 2025
- **[Google25 A2A]** Google et al., *Agent-to-Agent (A2A) Protocol* 2025