Exploring The Effects of LLM Hallucination On the Software Development Cycle

Nicholas Manning
College of Engineering and Science
Florida Institute of Technology
Melbourne, Florida, USA
Nmanning2024@my.fit.edu

Tyler Ton
College of Engineering and Science
Florida Institute of Technology
Melbourne, Florida, USA
TTON@my.fit.edu

Orion Powers
College of Engineering and Science
Florida Institute of Technology
Melbourne, Florida, USA
Opowers@my.fit.edu

***Abstract*—Large Language Models (LLMs) have become integral to software development workflows, yet their tendency to hallucinate, generating fluent but factually incorrect outputs poses systematic risks across the Software Development Lifecycle (SDLC). This paper presents a controlled three-arm experiment to quantify how hallucinations propagate through different development approaches. Using a Wordle API implementation as the experimental task, we compared three configurations: Arm A (human baseline), Arm B (single-agent LLM with end-to-end prompting), and Arm C (multi-agent pipeline with specialized PM, Engineer, and QA agents). Arm B achieved 100% functional correctness across all test cases, while Arm C consistently reached only 80%, with the 20% gap attributable to three tests asserting incorrect expected values. The root cause was a specification-level hallucination in the requirements phase that propagated faithfully through design and implementation artifacts due to context isolation between agents. The single-agent configuration detected and self-corrected this inconsistency, while the multi-agent pipeline's compartmentalized context prevented error detection. These findings demonstrate that context isolation in multi-agent SDLC pipelines can enable spec-level hallucination propagation that monolithic systems naturally mitigate, with direct implications for agentic software engineering tool design.**

***Index Terms*—Artificial Intelligence, LLM hallucination, Software Engineering development lifecycle, LLM integration**

I. Introduction

A. Motivation

The software development lifecycle is a topic that has remained in a constant state of evolution ever since its inception. Software Engineering arose as a means to tackle the human-driven processes required to optimize workflows and production. However, the introduction of Artificial Intelligence has upended many of the established aspects of how developers interact with their software systems[13]. There are already many understood risks and debts that are incurred throughout the development lifecycle, yet large language model integration has created an entirely new set of issues.

B. Problem Statement

Over the years, development methodologies have been introduced for software engineers working on projects. One common framework follows a seven-part series of: Planning, Requirements Analysis, System Design, Implementation, Testing, Deployment, and Maintenance. LLM usage — and subsequently their hallucinations — incurs a brand new dimension of complexity to consider when each step in the lifecycle is reached.

C. Research Gap

The knowledge gap stems from understanding how LLM usage specifically targets vulnerabilities within each stage of the lifecycle. The usage of the word "vulnerability" is multifaceted in this context. Vulnerabilities that derive from inherent human error are paired with those of the system. LLMs can save the user time, especially in situations where monotonous tasks are not automated. Hallucinations can take many forms as the breadth and width of the context for the LLM increases.

D. Paper Structure and Contributions

This paper evaluates the concept of hallucination from multiple dimensions. Firstly, understanding what research has already been conducted within the concept space is integral to establishing context and identifying potential knowledge gaps. Much of the existing knowledge on this topic focuses on the internal factors in LLMs that produce hallucinations. It is understood that the integration of LLMs can produce issues, but there is much less understood regarding their specific implications across each phase of the development lifecycle.

From there, the research sections are broken up into three arms. Arm A establishes a human baseline, assessing the developer's perspective when implementing the software manually without LLM assistance. Arm B implements a single-agent approach in which a complete specification is provided to a single LLM instance in one end-to-end prompt. Lastly, Arm C employs a multi-agent team composed of specialized sub-agents (Product Manager, Engineer, QA) working sequentially with isolated context to decompose the problem set. All three arms are evaluated against an identical test suite, providing a detailed comparative look into hallucination behavior and error propagation across the software lifecycle.

II. Background

The Software Development Lifecycle (SDLC) provides a structured framework for planning, creating, testing, and maintaining software systems. While various models exist — such as waterfall, agile, and spiral — they share a common set

of phases: planning, requirements analysis, system design, implementation, testing, deployment, and maintenance. Each phase produces artifacts and decisions that downstream phases depend on, meaning errors introduced early can propagate and compound across the entire lifecycle.

Large Language Models (LLMs) are a class of deep learning systems trained on massive amounts of text and code, enabling them to generate syntactically fluent and contextually relevant outputs in response to natural language prompts. Built on transformer architectures and refined through techniques such as reinforcement learning from human feedback (RLHF), modern LLMs have demonstrated strong performance across software engineering tasks including code generation, debugging, documentation, and test creation. Tools like GitHub Copilot and Amazon CodeWhisperer have brought these capabilities directly into developer workflows [14] [15].

Despite their utility, LLMs are probabilistic systems — they do not retrieve facts but generate statistically likely continuations of their input. This fundamental property gives rise to *hallucinations*: outputs that are fluent and syntactically plausible but factually incorrect, logically inconsistent, or unsupported by any ground truth[2]. Hallucinations are not random noise; they are coherent-sounding errors, which makes them particularly difficult to detect without deliberate verification[2][10]. When LLMs are integrated into an SDLC context, hallucinations cease to be isolated output errors and instead become risks that can infiltrate project artifacts, propagate across phases, and compound in ways that are costly to remediate.[6][12]

III. Related Work

In this section, we explore existing literature within the field to understand what work has been done toward the topic of LLM hallucinations. The sources are organized thematically to structure the discussion in further sections.

a) [2] Large Language Models Hallucination: A Comprehensive Survey [2].: Hallucination is defined as "the generation of content by an LLM that is fluent and syntactically correct, but factually inaccurate or unsupported by external evidence." The authors break hallucinations into two categories: *intrinsic hallucination*, where the LLM output contradicts the source material, and *extrinsic hallucinations*, which are "characterized by the inclusion of information that is not present in the ground truth."

The paper identifies where hallucinations can be detected across the LLM development lifecycle: (1) Data Curation, (2) Model Architecture, (3) Pre-training, (4) Fine-tuning, and (5) Inference.

b) [4] Hallucination is Inevitable: An Innate Limitation of Large Language Models [4].: This paper proves mathematically that hallucination cannot be eliminated from any LLM, regardless of architecture, training method, or prompting strategy. Using Cantor's diagonalization argument from learning theory, the authors prove three escalating theorems: (1) any enumerable set of LLMs will hallucinate on some inputs; (2) any enumerable set of LLMs will hallucinate on infinitely many inputs; (3) any individual computable LLM will inevitably hallucinate.

Since real-world LLMs are a subset of computable functions, the result applies universally. Retrieval-Augmented Generation (RAG) is identified as the most promising mitigation, since it provides information beyond training samples, but scalability remains an open problem.

c) [5] LLMs: A Game-Changer for Software Engineers? [5].: Haque (2025) examined the transformative role of LLMs in software engineering, arguing that while LLMs are genuine game-changers, their true potential is only realized when combined with human expertise. LLMs have demonstrated significant capabilities across code generation, debugging, testing, refactoring, and documentation, with productivity improvements reported at up to 55% for routine coding tasks. However, the study highlights critical limitations including lack of true code understanding, poor long-range context handling, high computational costs, and ethical concerns. The author concludes that LLMs should be viewed as augmentation tools rather than replacements for developers.

d) [6] LLM-Based Agents Suffer from Hallucinations: A Survey of Taxonomy, Methods, and Directions [6].: Lin et al. (2025) presented the first comprehensive survey on hallucinations in LLM-based agents, distinguishing them from conventional LLM hallucinations as more complex, multi-stage errors with real-world consequences. The authors proposed a novel taxonomy of five hallucination types: reasoning, execution, perception, memorization, and communication hallucinations. The survey identified eighteen triggering causes, and reviewed ten mitigation approaches grouped into three categories: knowledge utilization, paradigm improvement, and post-hoc verification.

e) [11] MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework [11].: Hong et al. (2024) introduce MetaGPT, a multi-agent framework that directly addresses the problem of cascading hallucinations in collaborative software engineering by encoding human-like Standardized Operating Procedures (SOPs) into agent workflows. MetaGPT structures multi-agent collaboration as an assembly line mirroring a real software company, assigning distinct roles (Product Manager, Architect, Engineer, QA Engineer) to different agents and requiring each to produce standardized deliverables. This reduces hallucination by transforming open-ended generation into structured completion. Despite these safeguards, the authors acknowledge that early implementations still overlooked errors during review due to LLM hallucinations, necessitating an executable feedback mechanism with up to three retries.

f) [3] Hallucination by Code Generation LLM?: Taxonomy, Benchmarks, Mitigation, and Challenges [3].: This paper expands on a taxonomy of hallucinations specific to the coding environment:

1) **Syntactic Hallucinations:** syntax violations; incomplete code generation
2) **Runtime Execution Hallucinations:** API knowledge conflicts; invalid reference errors

3) **Functional Correctness Hallucinations:** incorrect logical flow; requirements deviation
4) **Code Quality Hallucinations:** resource mishandling; security vulnerabilities; code smell

The paper also defines a list of root causes, including social bias, imitative falsehood, knowledge conflicts, domain knowledge deficiencies, outdated knowledge, attention mechanism limitations, positional encoding degradation, unidirectional contextualization, shortcut learning, overfitting, belief misalignment, inadequate evaluation metrics, and ambiguous input prompts.

g) [10] CodeMirage: Hallucinations in Code Generated by Large Language Models [10].: Agarwal and Pei (2025) present a taxonomy for the different types of coding defects that can be incurred by LLM hallucinations:

1) **Dead or Unreachable Code:** generated code has an unreachable or redundant piece of code.
2) **Syntactic Incorrectness:** generated code has syntactic errors and fails to compile.
3) **Logical Error:** generated code cannot solve the given problem correctly.
4) **Robustness Issue:** generated code fails on certain edge cases or raises an exception.
5) **Security Vulnerabilities:** generated code has security vulnerabilities or memory leaks.

h) [8] LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation [8].: Zhang et al. (2025) conducted an empirical study on LLM hallucinations within practical, repository-level code generation scenarios. The authors established a hallucination taxonomy organized into three overarching categories: (1) factual knowledge conflicts, (2) context-related conflicts, and (3) requirement violations. To mitigate these issues, the authors proposed a Retrieval-Augmented Generation (RAG) based method that constructs a retrieval library from the target repository and supplies relevant code snippets as supplemental prompts during generation.

i) [9] We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs [9].: Spracklen et al. (2025) presented a comprehensive analysis of package hallucinations, a specific form of code generation hallucination where LLMs generate code that references software packages that do not actually exist. Using 16 popular code-generating LLMs and two unique prompt datasets, the authors generated 576,000 code samples and found that 19.7% of generated packages were hallucinated across all tested models. Three mitigation strategies were evaluated: RAG, self-directed feedback, and supervised fine-tuning, with one model's rate dropping below 3%.

j) [1] Rapid End-to-End Test Generation and Hallucination Mitigation Using Generative Artificial Intelligence [1].: This paper assesses the impact that artificial intelligence has during the process of end-to-end testing on a cloud-based system. In traditional methods, this meticulous process is both time- and resource-intensive, often requiring full attention. Their approach does not simply dive into the direct use of AI; rather, it evaluates the pros and cons while also providing potential solutions to improve efficiency. The implications that hallucinations have on this testing process are widespread and highly variable. Within the process of automating test generation, the authors enumerate an error distribution:

1) Selector mismatches: 58%
2) Missing assertions: 25%
3) Redundant actions: 12%
4) Navigation breaks: 6%

The paper references Abstract class-based DOM referencing, single-shot prompting, prompt engineering optimization, and temperature tuning as potential mitigations.

k) [7] Who's in Charge? Disempowerment Patterns in Real-World LLM Usage [7].: Sharma et al. (2026) presented the first large-scale empirical analysis of disempowerment patterns in real-world AI assistant interactions, analyzing 1.5 million consumer Claude.ai conversations. The study identified three primary disempowerment mechanisms: reality distortion, value judgment distortion, and action distortion. The authors found that severe forms of disempowerment potential occur in fewer than one in a thousand conversations, though rates are substantially higher in personal domains compared to technical domains.

The relevant work section provides the necessary framework of understanding to build context for the methodology and experiments page. Much of what exists within this concept space is spread across a myriad of resources, so this serves as an organizational tool to provide precedent for future work.

IV. Methodology

A. System Overview and Hallucinations Across the Lifecycle

The implication of hallucinations varies across the different stages of the software development lifecycle. We follow the most common set of phases, with activities sourced from IBM [12].

1) Planning: During the planning phase, developers may use LLMs to assist with scoping, feasibility assessment, and resource estimation. Human oversight here is critical because hallucinations in this phase are foundational; a plausible but incorrect technology recommendation can propagate downstream into every subsequent phase. Detecting hallucinations at planning requires a developer to cross-reference LLM-generated recommendations against domain expertise, organizational context, and verified external sources. Human mitigation at this phase, therefore, requires treating all LLM output as a starting hypothesis to be validated, not a conclusion to be accepted.

2) Requirements Analysis: LLMs are increasingly used to assist with drafting and refining requirements documents. Hallucinations at this stage often manifest as requirements that sound technically sound but misrepresent stakeholder intent, introduce unspecified assumptions, or reference non-existent standards or regulatory frameworks. As noted by Alansari and Luqman [2], extrinsic hallucinations are especially difficult for

non-expert reviewers to catch. A human reviewer familiar with the project context is the primary defense at this stage.

3) System Design: At the system design phase, LLMs are used to suggest architectural patterns, propose component structures, and draft interface specifications. Hallucinations here can include references to design patterns that do not fit the described constraints or API designs that conflict with underlying frameworks. MetaGPT [11] demonstrated that encoding standardized operating procedures into agent workflows meaningfully reduces hallucination propagation between phases — a principle that translates directly to human review workflows as well.

4) Implementation: The implementation phase has received the most attention in hallucination research, as code generation is the most common LLM-assisted task in software development [5]. Lee et al. [3] provide a taxonomy of four hallucination categories specific to code generation: syntactic, runtime execution, functional correctness, and code quality hallucinations. Agarwal et al. [10] extend this with a complementary five-category defect taxonomy: dead or unreachable code, syntactic incorrectness, logical errors, robustness issues, and security vulnerabilities, which cuts across these phases.The optimal human oversight strategy is a complementary one: automated tools as a first pass for syntactic and structural violations, with human reviewers focused on semantic correctness, logical flow, and requirement fidelity.

5) Testing: During testing, LLMs are used to generate test cases, identify edge cases, and automate test scaffolding. Barathidass and Namburu [1] found that hallucinations in this context most commonly appear as selector mismatches, missing assertions, redundant actions, and navigation breaks. A key challenge is that a hallucinated test case may appear syntactically valid and even execute without error while failing to assert the correct behavior, creating a false coverage problem. This demands a deliberate human review step where test cases are traced back to requirements rather than simply executed.

6) Deployment: At deployment, LLMs may assist with generating configuration files, deployment scripts, infrastructure-as-code templates, and documentation. Hallucinations here can be particularly high-impact because errors in deployment artifacts can cause system outages, misconfigured security policies, or data exposure. Given that hallucination is mathematically inevitable regardless of model scale or prompting strategy [4], organizations cannot rely solely on improved models to eliminate this risk; procedural human checkpoints are a necessary architectural component of any LLM-augmented deployment workflow.

7) Maintenance: During maintenance, LLMs are used to assist with debugging, refactoring, and generating documentation for legacy codebases. Hallucinations in this phase often emerge as fabricated explanations of code behavior, incorrect identification of root causes, or suggested refactors that introduce new bugs. Human mitigation requires treating LLM-generated explanations as hypotheses to be tested empirically through targeted unit tests or logging, rather than as authoritative diagnoses.

B. Identifying Hallucinations

Detection is complicated by the defining characteristic of hallucinations: they are structured to look correct. Unlike traditional bugs that produce visible failures, hallucinations may pass validation, execute cleanly, and produce plausible results while being fundamentally wrong.

Automated detection works best for categories with verifiable ground truths. Syntactic and runtime hallucinations [3] can be caught by compilers and linters. Package hallucinations [9] can be flagged by validating imports against registries. However, automated detection fails at the semantic level. Functional correctness hallucinations and requirement deviations cannot be caught by tools that lack understanding of intended behavior. This gap motivates a layered detection strategy in which tools handle structural violations and human reviewers handle semantic defects — a division of labor that aligns with the breadth of categories documented in Agarwal et al. [10]'s defect taxonomy.

The practical implication is a complementary framework: automated tooling as the first pass for syntactic and structural hallucinations, with human reviewers focused on semantic correctness, requirement fidelity, and cross-phase consistency. As Haque [5] concludes, the human role in an LLM-augmented SDLC shifts from producing artifacts to critically auditing them.

V. Experiments

A. Experimental Setup

1) Purpose: Compare three SDLC operating modes — a human engineer, a single LLM session, and a multi-agent sub-team — on a simple coding task designed to surface hallucinations. The experiment produces artifacts at every SDLC phase (requirements, design, implementation, testing) so hallucinations can be scored per phase and tracked as they propagate.

2) Baseline Task: Wordle-Style Guessing API: A small REST API that runs a five-letter word-guessing game. A caller starts a game, submits up to six five-letter guesses, and the server returns per-tile feedback: *green* for a letter in the correct position, *yellow* for a letter in the word but in the wrong position, *gray* for a letter not in the word.

This task was chosen because:

- **Tiny surface:** four endpoints, one state object, one rule set; finishable by a human in two to three hours.
- **Concentrated hallucination signal:** one specific rule — duplicate-letter handling — that nearly every LLM gets wrong without prompting.
- **Clean acceptance testing:** 15 deterministic tests cover the rules, including the duplicate-letter edge cases.
- **Fast scoring:** every SDLC phase produces a short artifact, keeping the token budget small.

a) Rules (shared by all three arms)::

- The hidden target word is selected from a fixed 500-word list on game start.
- The player has six guesses; after the sixth, the game is over.
- Each guess must be a five-letter word from the same list; otherwise return `400` and do not consume a turn.
- Feedback rules: green = right letter, right spot; yellow = right letter, wrong spot; gray = letter not in word. Duplicate letters are matched greedily — green first, then yellow left-to-right — and each target letter can only "feed" one guess tile.
- The game ends when the player guesses the target or uses six guesses.

b) API Contract::

- `POST /games` → `201 { id, guesses_remaining: 6 }`
- `POST /games/:id/guess { word }` → `200 { feedback: [...], won, guesses_remaining }` or `400` if word is invalid or `409` if game is over
- `GET /games/:id` → `200 { id, state, guesses: [{word, feedback}], won, guesses_remaining }`
- `DELETE /games/:id` → `204`

c) Stack Constraints:: Node.js 20, Express 4.19, in-memory store (no DB). Validation: `zod 3.x`. Testing: `vitest 1.x` + `supertest 7.x`. The word list is provided as `words.json` (500 five-letter words).

3) The Three Arms:

a) Arm A: Human Engineer.: A single human developer implements the Wordle API from the specification without LLM assistance, serving as a baseline for comparison. The participant follows the same SDLC phases: requirements analysis, design, implementation, and testing—and is evaluated using the identical 15-test acceptance suite.

b) Arm B: Single-Agent LLM.: One continuous LLM session using Claude Code Sonnet 4.6. The experimenter supplies four prompts in order — requirements, design, implementation, testing — and pastes each prior response as context for the next prompt. No role label, no structured handoff schema. Same model, fixed temperature across all trials.

c) Arm C: Multi-Agent Sub-Team.: Four sub-agents using Claude Code Sonnet 4.6, each scoped to one phase, each producing a structured artifact that becomes the constrained input to the next:

- **PM (Project Manager):** Gathers all requirements
- **Architect:** API spec + data model (Design)
- **Engineer:** Source tree (Implementation)
- **QA:** Test plan + tests + report (Testing)

Same underlying model as Arm B. This isolates the effect of role specialization from the effect of model choice.

4) Execution Protocol:

- Run each arm three times for a pilot ($n = 3$) or five times ($n = 5$) for final results.
- Cap every trial at four hours of wall-clock time.
- Commit every intermediate artifact to a per-trial folder: `/trials/{arm}/{trial-id}/{phase}/...`
- Record: model version, temperature, seed, participant experience level, wall-clock time per phase.
- After all trials complete, run the shared 15-test acceptance suite against each implementation.

5) Scoring: Use the separately delivered Research Comparison Standard. Two raters classify every defect using the unified 12-category taxonomy (H1 through H12). Report hallucination count, density, phase distribution, propagation ratio, functional correctness (tests passed), wall-clock time, and Cohen's kappa per category.

For this Wordle task the three headline numbers to report are: (1) did duplicate-letter handling work — yes/no per trial per arm; (2) total hallucinations per arm aggregated across trials; (3) fraction of hallucinations that originated in an upstream artifact and survived into code.

6) Token and Time Budget:

- Arm A costs no tokens.
- Arm B per trial: $\sim$4 prompts, each $\sim$2k–4k tokens in + $\sim$2k–4k out. Three trials $\approx$ 50k tokens total.
- Arm C per trial: 4 sub-agents, each $\sim$3k in + $\sim$3k out. Three trials $\approx$ 70k tokens total.
- Scoring is done by humans, not by LLM, to keep the budget bounded and avoid using an LLM as judge.

7) Hypotheses:

- **H-a:** Arm B will produce more total hallucinations per artifact than Arm A.
- **H-b:** Arm C will reduce code-level hallucinations (H1–H5) relative to Arm B but will introduce agent-pipeline hallucinations (H10–H12) that Arm B cannot exhibit by construction.
- **H-c:** Duplicate-letter handling will fail in $\geq$50% of Arm B trials and in $<$50% of Arm C trials, because the Architect's written spec anchors the Engineer.

B. Results

1) Arm A: Human Engineer: The human engineer achieved a final score of 15/15 on the acceptance test suite after one fix cycle. The initial run produced 13/15, with the same three duplicate-letter edge-case tests (AT-11, AT-12, AT-13) failing. After debugging, the participant traced the algorithm by hand, identified that the task specification's expected values were incorrect, corrected the test assertions, and achieved 15/15.

TABLE I
ARM A ACCEPTANCE TEST RESULTS

Metric	Trial 1
Tests Passed (initial)	13 / 15
Tests Passed (after fix)	15 / 15
Fix Cycles	1
Duplicate-Letter Correct	Yes
ESM Patch Needed	Yes

Per-Trial Notes — Trial 1: The participant implemented the two-pass greedy coloring algorithm directly from the specification's rule description and produced a working API with no structural issues. On the first test run, 13 of 15 tests passed; AT-11, AT-12, and AT-13 failed.

The participant initially suspected a bug in the coloring algorithm. To debug, the participant hand-traced the two-pass algorithm on the `APPLE`/`PAPER` example and found that `PAPER[2]=P` matches `APPLE[2]=P`, which produces a green in Pass 1. The specification's stated expected value of gray at position 2 directly contradicts the greens-first rule described in the same specification. The participant then verified the `PIPER` and `SEEDY` examples, finding analogous errors in each.

After confirming the algorithm implementation was correct, the participant updated the three test assertions to match the correct algorithm output and reran the suite. All 15 tests passed.

One additional issue was encountered during test authoring: the initial attempt to override the random word selection used a direct ES module named-export reassignment, which is disallowed in ESM. The fix was to refactor `wordList.js` to export a mutable `wordListApi` object whose `randomWord()` method can be swapped at test time. This is the same ESM patching issue encountered in Arm B Trial 1.

TABLE II
ARM A PER-TEST RESULTS BEFORE AND AFTER THE FIX CYCLE

Test ID	Description	Initial	After Fix
AT-01	Create new game	PASS	PASS
AT-02	Correct guess wins	PASS	PASS
AT-03	Invalid word → 400	PASS	PASS
AT-04	Guess after win → 409	PASS	PASS
AT-05	GET game state	PASS	PASS
AT-06	DELETE game	PASS	PASS
AT-07	Nonexistent game → 404	PASS	PASS
AT-08	All gray feedback	PASS	PASS
AT-09	All green feedback	PASS	PASS
AT-10	Mixed feedback	PASS	PASS
AT-11	Dup. letter: APPLE vs PAPER	FAIL	PASS
AT-12	Dup. letter: APPLE vs PIPER	FAIL	PASS
AT-13	Green priority: SPEED vs SEEDY	FAIL	PASS
AT-14	Six wrong → lost	PASS	PASS
AT-15	Guess on lost → 409	PASS	PASS

TABLE III
ARM B ACCEPTANCE TEST SCORES

Metric	Trial 1	Trial 2	Trial 3
Tests Passed	15/15	15/15	15/15
Tests Failed	0	0	0
Fixes Required	Yes (1)	Yes (1)	None
Score (%)	100	100	100

2) Arm B: Single-Agent LLM: Trial 1. One fix was required: the test file used a direct ES module named-export mutation to override `randomWord()`, which is disallowed in ESM. Fix: `wordList.js` was updated to export a mutable `wordListApi` object, and `routes/games.js` was updated to call `wordListApi.randomWord()`. All 15 tests passed after this change.

Trial 2. One fix was required: the generated test suite referenced words (`BLISS`, `FLUFF`, `JOKER`) not present in the shared `words.json`. These three words were added. All 15 tests passed after this change. The `wordListApi` pattern was correctly used from the start.

Trial 3. No fixes required. The implementation and test suite were correct as generated. All 15 tests passed on the first run.

TABLE IV
SUMMARY OF ARM B EXPERIMENTS

Arm B — Multi-Turn (full prior context)	
Trials Completed	3
Mean Score	15/15 (100%)
Trials Requiring Fixes	2 of 3 (minor)

Arm B achieved a perfect score across all three trials. The multi-turn prompting strategy, where each phase received complete prior context, produced consistent, correct, and runnable implementations. Minor fixes were needed in two trials — one for ESM module patching and one for word list coverage — neither of which indicates a fundamental design flaw.

3) Arm C: Multi-Agent Sub-Team: Arm C evaluates a MetaGPT-style multi-agent pipeline in which four specialized agents — Project Manager (PM), Architect, Engineer, and QA Engineer — each operate with strict information isolation. Unlike Arm B, each Arm C agent sees only the immediate upstream artifact, never the original task specification or downstream artifacts. This isolation is the experimental manipulation and the primary source of hallucination risk in this arm.

Three independent trials were run. Each trial produced: `REQS.md` (PM), `API_SPEC.md` + `DATA_MODEL.md` (Architect), full source tree (Engineer), and `tests/acceptance.test.js` + `TEST_REPORT.md` (QA).

a) Pipeline Structure and Information Flow.: The four-agent chain operates as follows, where each arrow represents the only permitted information flow between phases:

Task Spec → [PM] → `REQS.md` → [Architect] → `API_SPEC.md` + `DATA_MODEL.md` → [Engineer] → *Source Code*

`REQS.md` + *Source Code* → [QA] → `tests/acceptance.test.js` + `TEST_REPORT.md`

The QA agent receives `REQS.md` and source code, but *not* the Architect's design documents. Any drift between `REQS.md` and the design docs is invisible to QA and can only be detected through test failures.

All three trials produced an identical result: 12/15 acceptance tests passing, with the same three tests (AT-11, AT-12, AT-13) failing in every trial. This consistency across

TABLE V
ARM C ACCEPTANCE TEST SCORES

Metric	Trial 1	Trial 2	Trial 3	Mean
Tests Passed	12/15	12/15	12/15	12.0/15
Tests Failed	3	3	3	3.0
Score (%)	80	80	80	80
Fixes Required	None	None	None	—

independent runs indicates the failures are systematic, not stochastic.

TABLE VI
CROSS-ARM PERFORMANCE COMPARISON

Arm	Strategy	Score (%)	Consistent Failures
A	Human baseline, no LLM	100	None
B	Single-agent, full context	100	None
C	Multi-agent, isolated context	80	AT-11, AT-12, AT-13

b) Root Cause: Spec Hallucination Propagation.: The failures share a single root cause: the original task specification contains three algorithmically incorrect edge-case examples. Each example omits or misplaces a green tile that the standard two-pass greedy algorithm correctly produces:

- **AT-11 / AT-12** (target=`APPLE`): The spec treats `APPLE` as having one P, but it has two. Any guess with P at position 2 will match `APPLE[2]=P` and receive green in pass 1. The spec examples omit this green.
- **AT-12** (guess=`PIPER`): The spec claims position 3 (E) is green, but `APPLE[3]=L` $\neq$ `E`. No correct implementation can produce green at position 3 for this guess.
- **AT-13** (target=`SPEED`, guess=`SEEDY`): `SPEED[2]=E` and `SEEDY[2]=E` are identical — pass 1 must award green. The spec states gray at position 2, contradicting its own greens-first rule.

These errors propagated through the entire pipeline unchanged:

- The PM agent faithfully transcribed the spec's stated expected values into `REQS.md` without independent verification.
- The Architect, seeing only `REQS.md`, encoded the incorrect expected values into the algorithm specification and error catalog.
- Despite the incorrect examples in the design docs, the Engineer implemented the *correct* two-pass algorithm, as the descriptive algorithm text was consistent. The implementation did not match the examples — it followed the rule.
- The QA agent, seeing `REQS.md` and source code, wrote tests asserting the `REQS.md` expected values. These tests correctly describe the spec, but the spec is wrong — so the tests fail against a correct implementation.

This creates the defining finding of Arm C: *the implementation is algorithmically correct, the tests are specification-faithful, and they are in direct contradiction because the specification itself contains hallucinated examples.*

c) Propagation Trace.: The spec error exhibits the longest propagation chain observed in this experiment:

Task Spec (H-origin) $\rightarrow$ `REQS.md` (H-1) $\rightarrow$ `API_SPEC.md` (H-2) $\rightarrow$ `tests/acceptance.test.js` (H-3)

The Engineer broke the chain by implementing the correct algorithm, but because the QA agent was isolated from the design docs, it could not detect the contradiction. The propagation count for this hallucination is 3.

d) What Arm B Did Differently.: Arm B encountered the same incorrect spec examples but identified them as errors and documented the discrepancy explicitly in its `TEST_REPORT.md`. The single-agent advantage here is not intelligence — it is *continuity of context*. The agent could reason across the spec text, the algorithm definition, and the examples simultaneously, detecting the inconsistency. In Arm C, no single agent had visibility across all three layers simultaneously.

VI. DISCUSSION

A. Arm A: Human Engineer Discussion

a) Human Detection of Specification Errors.: The human participant's debugging process demonstrates a critical advantage of human cognition in SDLC workflows: the ability to validate specification consistency through first-principles reasoning. When the initial test run produced three failures (AT-11, AT-12, AT-13), the participant did not accept the specification's expected values as ground truth. Instead, the participant hand-traced the two-pass greedy algorithm on paper, step by step, to verify the implementation's correctness. This manual simulation revealed that the specification's edge-case examples contradicted its own algorithmic description—a discrepancy that required cross-referencing multiple sections of the spec simultaneously.

This detection pattern is significant because it illustrates a form of error-checking that neither Arm B nor Arm C employed reliably. The human participant treated the specification as a *hypothesis to be validated* rather than an authoritative input, applying domain knowledge of how greedy matching algorithms behave. This meta-cognitive skepticism—questioning whether the specification itself could be wrong—is not a capability that LLMs demonstrate consistently, even when they have access to contradictory information.

b) Fix Strategy: Correcting the Ground Truth.: Unlike both LLM-based arms, which attempted to reconcile specification inconsistencies through code modifications or additional logic, the human participant chose to *fix the tests* rather than the implementation. After confirming the algorithm was correct, the participant updated the three failing test assertions to match the algorithmically correct output. This represents a fundamentally different error resolution strategy: recognizing that the acceptance criteria themselves were flawed.

This approach carries important implications for LLM-augmented development. If an LLM is prompted to "fix the failing tests," it may modify the implementation to match incorrect expected values rather than questioning whether the

expected values are wrong. The human participant's willingness to reject the specification's authority and redefine the ground truth demonstrates a form of critical reasoning that current LLM systems do not reliably exhibit.

c) The ESM Module Patching Challenge.: The human participant encountered the same ES module export mutability issue as Arm B Trial 1: attempting to monkey-patch a named export for test stubbing, which is disallowed in ESM. The participant resolved this by refactoring `wordList.js` to export a mutable object with a `randomWord()` method, enabling test-time substitution without violating module semantics. This is identical to the solution Arm B converged on, suggesting that this particular technical challenge is not specific to LLM-based workflows but rather an inherent property of the ESM standard when combined with test mocking requirements.

d) Baseline Performance and Error Rate.: Arm A achieved 100% functional correctness after one fix cycle, with zero hallucinations in the implementation artifacts. The only errors present were in the *task specification itself*, which the participant identified and corrected. This establishes an important baseline: a competent human developer, working without LLM assistance, produced a fully correct implementation and detected specification-level inconsistencies that propagated unchecked through the multi-agent pipeline in Arm C. The human baseline's perfect post-fix correctness rate (15/15) provides a reference point against which to evaluate both single-agent (Arm B: 15/15) and multi-agent (Arm C: 12/15) LLM performance.

B. Arm B: Single-Agent LLM Discussion

a) Spec Example Discrepancy.: The original task specification contained an incorrect edge-case example:

- **Stated:** target=`APPLE`, guess=`PAPER` → [yellow, yellow, gray, gray, gray]
- **Correct:** target=`APPLE`, guess=`PAPER` → [yellow, yellow, green, yellow, gray]

`APPLE` contains two P's at positions 1 and 2 (A-P-P-L-E). `PAPER[2]=P` aligns exactly with `APPLE[2]=P`, so the two-pass algorithm correctly marks it green in pass 1. All three trials identified this discrepancy and wrote tests asserting the algorithmically correct result.

b) ESM Module Patching.: A recurring challenge was overriding the random word selection in tests. Named ES module exports are read-only and cannot be monkey-patched. The consistent solution across all trials was to export `randomWord()` as a method on a mutable object (`wordListApi`), allowing tests to substitute the method without violating ESM semantics.

c) Word List Coverage.: Trial 2's generated tests referenced words not present in the shared `words.json`. A more robust approach would be to select the wrong guess words dynamically from the loaded word list.

C. Arm C: Multi-Agent Sub-Team Discussion

a) Specification Hallucinations as a Unique Risk in Multi-Agent Systems.: The Arm C failures demonstrate a hallucination failure mode that is amplified by — and in some respects unique to — multi-agent pipelines. In a single-agent system, an inconsistency within the input specification is visible to the agent as a single reasoning problem. In a multi-agent system, the inconsistency is split across context boundaries: one agent sees the rule, a second sees the examples, a third tests the output. No agent sees all three simultaneously, so no agent can detect the contradiction.

This is consistent with Lin et al. [6]'s category of *communication hallucination*: the downstream QA agent received a signal (`REQS.md`) That was inconsistent with the upstream engineering output, but lacked the context to recognize the source of the inconsistency.

b) The Engineer's Partial Correction.: A notable finding is that the Engineer agent implemented the correct algorithm in all three trials despite receiving design documents that encoded incorrect expected values. This suggests the Engineer's algorithm was driven by the descriptive pseudocode ("green first, then yellow left-to-right") rather than the worked examples — a form of specification-reading that privileged procedural text over illustrative output. However, this partial correction was invisible to the QA agent, which had no access to the design documents.

c) Implications for Pipeline Design.: These findings suggest several concrete mitigations for multi-agent SDLC pipelines:

- **Cross-phase review:** Allow QA agents read access to design documents, not only requirements, to detect implementation-spec drift.
- **Consistency checking:** Include an explicit "consistency verification" agent that compares worked examples in requirements against the algorithmic definition, before the Engineer phase begins.
- **Example-driven TDD:** Have the Engineer write and run unit tests for the stated edge cases before writing the full implementation, surfacing spec errors earlier in the pipeline.
- **Specification grounding:** Require the PM agent to derive worked examples programmatically from the stated algorithm rather than transcribing them from the raw spec.

d) Limitations.: This experiment used a single task (Wordle API) and a single model (`claude-sonnet-4-6` at temperature 0). Results may not generalize to more complex tasks, different model families, or non-zero temperatures. The three-trial sample is sufficient to establish consistency of the hallucination , but insufficient for statistical inference about variance.

e) Summary.: Arm C consistently achieved 80% functional correctness. The 20% gap relative to Arm B is entirely attributable to three tests that assert incorrect expected values derived from a hallucinated edge-case specification. The multi-agent The pipeline faithfully propagated the specification error through three artifact layers, while the single-agent pipeline detected and corrected it. This result provides empirical support for the hypothesis that context isolation in Multi-

agent SDLC pipelines increase hallucination propagation risk, particularly for specification-level inconsistencies.

VII. Threats to Validity

A. Internal Validity

The experiment uses a single task (Wordle API), a single model family (Claude Sonnet 4.6), and a small sample size ($n = 3$ trials per arm for Arms B and C, $n = 1$ for Arm A). This limits causal inference in several ways. First, the consistency of the hallucination pattern across Arm C trials (3/3 failures on AT-11, AT-12, AT-13) establishes replicability within this configuration but does not rule out task-specific effects. A different coding challenge—one without a concentrated edge-case vulnerability—might not exhibit the same propagation behavior. Second, the use of a single model at temperature 0 controls for model-specific variance but prevents generalization to other LLM families (GPT-4, Gemini, Llama) or non-deterministic sampling strategies. Different models may exhibit different specification-reading behaviors; for instance, a model with stronger emphasis on example-driven reasoning might propagate worked examples more faithfully than procedural descriptions, potentially exacerbating or mitigating the observed effect.

The low trial count ($n = 3$) is sufficient to demonstrate consistency of the hallucination pattern but insufficient for statistical power or confidence interval estimation. We report the observed behavior as a repeatable phenomenon rather than a statistically validated effect size. Future work should increase n and diversify task complexity to strengthen causal claims.

B. Construct Validity

The paper references a 12-category hallucination taxonomy (H1–H12) in the methodology section but does not systematically apply it in the results analysis. This creates a gap between the proposed measurement framework and the actual evaluation conducted. The taxonomy—derived from Lee et al. [3] and Agarwal et al. [10]—encompasses syntactic errors, runtime failures, logic bugs, dead code, security vulnerabilities, package hallucinations, API misuse, specification deviations, test assertion errors, role confusion, context loss, and communication hallucinations. However, the experiment focuses primarily on one category: specification-level hallucinations that propagate into test assertions (H9, H12).

This narrow focus raises two construct validity concerns. First, we cannot claim the taxonomy captures *all* relevant hallucination types without empirical validation across diverse tasks and failure modes. The Wordle API task surfaced specification propagation failures but may not exercise categories like package hallucination (H6) or security vulnerabilities (H5). Second, the absence of inter-rater reliability metrics (e.g., Cohen's kappa) means we cannot verify that the taxonomy's categories are consistently interpretable or that independent raters would classify the same defects identically. Future work should apply the full taxonomy to a broader task set and report agreement statistics to validate the classification scheme.

C. External Validity

The Wordle API is a deliberately simplified task: 500 lines of code, four REST endpoints, one core algorithm, and 15 acceptance tests. This enables controlled comparison but limits generalizability to real-world software development contexts. Several factors constrain external validity:

- **Task complexity:** Enterprise codebases involve thousands of files, cross-module dependencies, integration points, and evolving requirements. The specification propagation failure observed in Arm C may scale differently in systems where QA agents must synthesize test cases from dozens of design documents rather than a single `REQS.md` file.
- **Domain specificity:** The Wordle API is a self-contained algorithmic task with no external dependencies, database schemas, or third-party APIs. Real-world SDLC phases involve API integrations, compliance requirements, and deployment configurations—contexts where hallucination types not exercised here (e.g., package hallucination, configuration drift) may dominate.
- **Time constraints:** The two- to four-hour trial duration does not reflect multi-week sprint cycles, iterative requirement refinement, or long-term maintenance phases where hallucination accumulation and detection patterns may differ.

The findings demonstrate a *proof of concept* that context isolation in multi-agent pipelines can enable specification-level error propagation. Whether this effect persists, amplifies, or diminishes in production-scale development workflows remains an open empirical question.

D. Human Participant Bias

Arm A involved a single human participant implementing the Wordle API without LLM assistance. Several sources of bias may have influenced the results:

- **Awareness of evaluation criteria:** The participant knew the implementation would be evaluated using a 15-test acceptance suite and that duplicate-letter handling was a focal point. This foreknowledge may have increased vigilance during debugging, leading to more deliberate validation of edge-case examples than would occur in naturalistic development settings.
- **Participant expertise:** The participant's prior experience with algorithmic problem-solving and testing may not represent typical developer performance. A less experienced developer might not have hand-traced the algorithm or questioned the specification's authority, potentially accepting the incorrect expected values as ground truth.
- **Absence of replication:** With $n = 1$ for Arm A, we cannot assess inter-participant variance. The observed behavior—detecting spec errors through manual simulation—may be participant-specific rather than a general property of human cognition in SDLC contexts.

Future work should replicate Arm A with multiple participants across varying experience levels and blind them to the specific test cases to reduce demand characteristics.

E. Rater Reliability

The methodology section references a two-rater hallucination classification process using the H1–H12 taxonomy and mentions Cohen's kappa as the inter-rater agreement metric. However, no kappa values are reported in the results. This omission undermines confidence in the defect classification process. Without reliability statistics, we cannot determine whether the absence of reported hallucinations in certain categories (e.g., H6: package hallucination, H10: role confusion) reflects genuine absence or rater disagreement about categorization.

Cohen's kappa values should be computed for each taxonomy category and reported in the results section. Values $\kappa \geq 0.80$ would indicate strong agreement; values $0.60 \leq \kappa < 0.80$ would suggest moderate agreement requiring adjudication; values $\kappa < 0.60$ would indicate the category definitions need refinement. The lack of these metrics in the current draft represents an incomplete validation of the measurement instrument.

VIII. Conclusion

This paper presented a controlled three-arm experiment comparing hallucination susceptibility and error propagation across human-driven development (Arm A), single-agent LLM assistance (Arm B), and multi-agent SDLC pipelines (Arm C). Using a Wordle API implementation as the experimental task, we evaluated each configuration against an identical 15-test acceptance suite. Arm A (human baseline) and Arm B (single-agent) both achieved 100% functional correctness, while Arm C (multi-agent) consistently reached only 80%, with the 20% gap attributable to three test cases asserting incorrect expected values derived from a hallucinated specification. The core finding is that *context isolation* in multi-agent pipelines enables specification-level hallucinations to propagate faithfully through requirements, design, and testing phases without detection, while single-agent systems operate with continuous context identify and self-correct the same inconsistencies. This result challenges the assumption that specialized agent roles inherently improve LLM-augmented development workflows and demonstrates that compartmentalized context can amplify certain hallucination failure modes rather than mitigate them.

The practical implications for agentic software engineering tool design are concrete. Multi-agent SDLC frameworks should implement cross-phase review mechanisms, allowing downstream agents (e.g., QA) read access to upstream artifacts (e.g., design documents) to detect implementation-specification drift. Consistency verification agents that validate worked examples against algorithmic definitions before implementation begins can surface specification errors earlier in the pipeline. Example-driven test development, where engineers write unit tests for stated edge cases before full implementation, provides an additional checkpoint for detecting spec-level inconsistencies. Finally, requiring product management agents to derive worked examples programmatically from algorithmic descriptions rather than transcribing them from raw specifications can prevent hallucinated examples from entering the artifact chain. These mitigations directly address the context isolation problem demonstrated in this experiment.

Future work should extend this investigation across multiple dimensions. First, task diversity: replicating the experiment with larger, more complex codebases, spanning database integration, API orchestration, and deployment configuration will test whether the observed propagation behavior scales or attenuates in real-world SDLC contexts. Second, model diversity: comparing results across LLM families (GPT-4, Gemini, Llama) and architectures will determine whether specification-reading strategies (privileging procedural text vs. worked examples) vary systematically by model. Third, sampling strategies: testing non-zero temperature settings will reveal whether stochastic generation introduces additional hallucination modes or disrupts the consistency observed at temperature 0. Fourth, statistical power: increasing sample size ($n \geq 10$ per arm) will enable hypothesis testing with confidence intervals rather than descriptive pattern reporting. Finally, longitudinal studies tracking hallucination accumulation and detection across multi-sprint development cycles will capture error propagation dynamics absent in single-pass implementations. These extensions will clarify the boundary conditions under which context isolation amplifies hallucination risk and inform the design of robust LLM-augmented development tools.

Acknowledgment

References

[1] C. Barathidass and K. Namburu, "Rapid end-to-end test generation and hallucination mitigation using generative artificial intelligence," *IEEE Access*, vol. 14, pp. 20605–20619, 2026. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=11363193

[2] A. Alansari and H. Luqman, "Large Language Models Hallucination: A Comprehensive Survey," arXiv, preprint arXiv:2510.06265, Oct. 2025. [Online]. Available: https://arxiv.org/abs/2510.06265

[3] Y. Lee, J. Y. Song, D. Kim, J. Kim, M. Kim, and J. Nam, "Hallucination by Code Generation LLMs: Taxonomy, Benchmarks, Mitigation, and Challenges," arXiv, preprint arXiv:2504.20799, Apr. 2025. [Online]. Available: https://arxiv.org/abs/2504.20799

[4] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is Inevitable: An Innate Limitation of Large Language Models," arXiv, preprint arXiv:2401.11817, Jan. 2024. [Online]. Available: https://arxiv.org/abs/2401.11817

[5] Md. A. Haque, "LLMs: A Game-Changer for Software Engineers?" *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 5, Article no. 100204, 2025, doi: 10.1016/j.tbench.2025.100204. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772485925000171

[6] X. Lin et al., "LLM-Based Agents Suffer from Hallucinations: A Survey of Taxonomy, Methods, and Directions," arXiv, preprint arXiv:2509.18970, Sep. 2025. [Online]. Available: https://arxiv.org/abs/2509.18970

[7] M. Sharma, M. McCain, R. Douglas, and D. Duvenaud, "Who's in Charge? Disempowerment Patterns in Real-World LLM Usage," arXiv, preprint arXiv:2601.19062, Jan. 2026. [Online]. Available: https://arxiv.org/abs/2601.19062

[8] Z. Zhang et al., "LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, Issue ISSTA, Article no. ISSTA022, Jul. 2025, doi: 10.1145/3728894. [Online]. Available: https://arxiv.org/abs/2409.20550

[9] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," in *Proc. 34th USENIX Security Symposium (USENIX Security '25)*, Seattle, WA, USA, Aug. 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen

[10] V. Agarwal, Y. Pei, S. Alamir, and X. Liu, "CodeMirage: Hallucinations in Code Generated by Large Language Models," arXiv preprint arXiv:2408.08333, Aug. 2024. [Online]. Available: https://arxiv.org/abs/2408.08333

[11] S. Hong et al., "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2024. [Online]. Available: https://arxiv.org/abs/2308.00352

[12] IBM, "What is the software development lifecycle?" [Online]. Available: https://www.ibm.com/think/topics/sdlc

[13] M. Mohamed, M. Assi, and M. Guizani, "The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review," arXiv preprint arXiv:2507.03156, 2025. [Online]. Available: https://arxiv.org/abs/2507.03156

[14] GitHub, "GitHub Copilot: Your AI pair programmer," [Online]. Available: https://github.com/features/copilot

[15] Amazon Web Services, "Amazon CodeWhisperer," [Online]. Available: https://aws.amazon.com/codewhisperer/