# **Technical Research Report: Mitigating Generative Diligence Degradation in Iterative Development**

Version: 24.0

Date: 18 June 2025

Status: Approved

1. Abstract

This report addresses a critical challenge observed during the iterative development of our agentic system: Generative Diligence Degradation (GDD). This phenomenon occurs when an LLM, tasked with incrementally modifying complex artifacts over a long interaction, inadvertently drops or over-simplifies existing, stable features that are not the focus of the immediate change request. Our experience with the evolution from supervisor\_v6.py to v7.py serves as a prime case study. The solution is not to demand greater "memory" from the LLM, but to architect a system that programmatically enforces software engineering best practices. This report outlines three core practices—Formal Structured Specifications, Version Control, and Test-Driven Development—and explains how our multi-agent pipeline is being explicitly designed to implement them, thereby transforming the LLM from a fallible collaborator into a component within a robust, self-regulating development framework.

2. The Problem: Generative Diligence Degradation (GDD)

During the generation of supervisor\_v7.py, a key feature from v6—the user feedback loop in the final review stage—was omitted. This was not a logical error, but a degradation of diligence. The LLM, focused on the complex new task of implementing the Coding Agent, lost the context of a less-central, existing feature.

This GDD is an inherent risk in any long-term, iterative development process that relies on an LLM's conversational context. The model's attention is finite and biased toward the most recent instruction. Relying on the LLM to "remember" all existing features while implementing new ones is fragile and unreliable.

3. Best Practices for Mitigation

The solution is to build a system around the LLM that enforces discipline. We have identified three industry-standard best practices that directly mitigate GDD.

* 1. Formal, Structured Specifications: The most reliable way to prevent feature loss is to provide the agent with an explicit, machine-readable checklist of *all* required functionalities for a given file. The agent's output must be validated against this specification.
* 2. Version Control: A robust version control system (like Git) provides a safety net. It allows for changes to be reviewed as discrete diffs, making regressions immediately obvious. It also provides a complete, traceable history and the ability to revert faulty changes.
* 3. Test-Driven Development (TDD): A comprehensive suite of automated tests acts as the ultimate quality gate. If a change, however small, breaks an existing feature covered by a test, the build fails. This programmatically prevents regressions.

4. How Our Architecture Implements These Practices

The autonomous R&D pipeline we are building is the direct implementation of these best practices, using agents to enforce the discipline.

* Formal Specs via design\_synthesis.json: Our pipeline's design\_synthesis.json artifact *is* the formal, structured specification. Instead of asking the Coding Agent to "add a feature," the Master PM and Design Agents will produce a JSON work order that lists *all* required functions, classes, and UI elements for a given file. The Coding Agent's job is to produce code that satisfies this structured manifest.
* Version Control via a Future Tooling Agent: The next evolution of our Coding Agent will not use a simple write\_file tool. It will be given tools like git\_create\_branch, git\_commit, and git\_create\_pull\_request. The commit messages will be programmatically generated from the high\_level\_goal in the design JSON, creating a perfectly documented history. Human review then happens at the pull request level, the industry standard.
* TDD via an Enhanced Coding Loop: The Coding Agent's Plan -> Execute loop will be extended to Plan -> Execute -> Test. The design\_synthesis.json will contain a tests\_to\_run key. After writing the code, the agent will be required to use a run\_tests tool. If the tests fail, it will be forced to loop back, read the error, and attempt to fix its own code, preventing regressions before a pull request is ever created.

5. Conclusion

The problem of "coding diligence" is not a flaw to be fixed with better prompts alone; it is a fundamental challenge to be solved with better architecture. By building an agentic system that programmatically enforces the core disciplines of modern software development, we mitigate the risk of GDD. The LLM becomes a powerful, creative engine for generating code, but it operates within a larger, more rigorous framework that provides the structure, memory, and quality assurance that ensures the project's long-term stability and success.