ChatGPT

Definition of Agentic Behavior

Agentic AI code assistants are systems that don't just generate code in isolation – they actively **plan, act, and adapt** over a sequence of steps towards a goal. In this context, *agentic behavior* means the assistant maintains a **sense of purpose and continuity** throughout a coding task, rather than behaving like a one-off code generator. Key aspects include:

- **Autonomy in Planning and Actions:** The agent can interpret an open-ended software task and break it into steps, deciding what code to write, edit, test, or delete next without constant human prompts [1] [2]. It proactively seeks information (reading files, documentation) and invokes tools (compilers, linters, tests) as needed [3]. This contrasts with a passive "autocomplete" that only responds turn by turn.

- **Architectural Awareness:** The agent understands the **structure of the codebase and its design architecture**, not just the snippet at hand. For example, it knows which module or layer it's working in (UI vs. core logic, etc.) and the role/purpose of each component in that architecture. The *Standard Model of Code* provides a formal way to encode this: every code element has dimensions like **Layer (D2)** – e.g. Interface, Application, Core – and **Role (D3)** – e.g. Controller, Repository – describing its place in the system [4] [5]. An agentic assistant leverages this context; it treats a function or class not as an isolated block, but as a "*holon*" in a larger system (a whole that is part of bigger wholes) [6] [7]. For instance, if modifying a *Controller* in the Presentation layer, an agentic assistant remains mindful of how that change impacts the Service or Database layers.

- **Task Continuity and Context Retention:** Rather than forgetting what it was doing after each response, the agent carries an **ongoing memory of the session state** – the goals, decisions made, code written, and feedback received. It exhibits *persistent state* across turns, maintaining coherence in its strategy [8]. For example, if the plan is to implement a feature through several files, the agent keeps track of which sub-tasks are done and what remains. It doesn't "reset" arbitrarily or propose redundant changes. This requires overcoming the inherent short-term memory limits of an LLM (fixed context windows) by using extended context or external memory. Modern agent frameworks address this via high-context models (e.g. Gemini with 1 million tokens context [9]) or persistent memory stores (e.g. vector databases) to retain facts beyond the immediate prompt [10] [11].

- **Deletion Awareness and Revision Tracking:** An agentic code assistant is mindful of changes it makes – including code it *removed*. A non-agentic agent might delete a function in one step and later re-introduce it erroneously because it forgot the deletion. Agentic behavior means keeping an **accurate mental model of the codebase state**. If a function `foo()` was removed, the agent remembers that and won't call `foo()` later or resurrect its logic without reason. This ties into having a reliable *virtual workspace model* of the project. Some advanced tools give the agent a project tree or state to update as it goes, so it knows at each step what the "world" looks like. Additionally, **version control awareness** can help – understanding diffs and commit history to avoid backtracking on previous deletions or edits.

- **Alignment with Purpose and Role (D3, D7):** Agentic assistants align their actions with the *intended purpose* of the code and the *role* it plays in the software. In the Standard Model, **D3 (Role)** represents the semantic purpose of a code "particle" (e.g. is this function a data Loader, a Validator, a Controller?) [12]. An agent should preserve that: if a function's role is "Validator", the agent shouldn't turn it into a data exporter during a refactor. Likewise, **D7** often corresponds to how and *when* something is used – in Standard Model v2 this is captured as **Lifecycle/Activation** (e.g. is this component created at startup vs. on-demand, does it react to an event?) [13]. An agentic assistant remains aware of these contextual dimensions. We can think of D3 as the *local purpose* (the function's job) and D7 as the *temporal/trigger context* for that code. Keeping alignment means *the agent's changes should not drift from the original purpose or violate the expected usage pattern*. For example, if editing a function meant to be called on a schedule (say a cron job, Activation=Time), it shouldn't refactor it in a way that only works for one-off invocation unless asked. This is related to the lens of **Semantics** (Lens R7 in Standard Model), which asks "What does it mean? What is the intent?" [14] [15]. An agentic assistant constantly checks that its outputs still fulfill the intended meaning and intent of the code it's modifying, rather than just achieving superficial correctness.

In summary, *agentic behavior* in code assistants is about **treating the development session as an ongoing autonomous process** – the AI is aware of itself and the code in a multi-step loop: it perceives the environment (codebase, tests, user instructions), maintains an inner state of "what we're doing and why", and produces actions (code edits, run commands, explanations) that continuously align with the project's architectural context and the user's goals [3] [16]. This stands in contrast to "fragmentary" behavior where the AI would just respond to the last prompt in isolation (often leading to inconsistency or losing the plot).

Root Causes of Behavioral Collapse in Agents

Despite a strong initial prompt (e.g. "You are an expert software agent, follow the plan, etc."), many AI agents **regress to shallow or fragmentary behavior mid-session**. They may start ignoring the plan, forgetting context, or reverting to generic responses. Several root causes contribute to this mid-session collapse:

- **Context Drift and Memory Limitations:** As conversations grow, the model's earlier instructions and architectural understanding may literally fall out of the prompt window or get "compressed". LLMs have finite context lengths, and many coding agents historically used only a few thousand tokens. Once the session exceeds that, older details (including the initial system role or the agent's own plan) may be dropped or summarized, which can dilute critical details. Even within the context, *model attention tends to skew toward the most recent content.* Research on *context drift* shows that an LLM's responses can undergo a *"slow erosion of intent"* – they gradually diverge from the originally specified instructions as the turns progress [17] [18]. For example, a code assistant that was initially told to use a certain coding style or follow a particular high-level approach might slowly stop doing so 20 messages later. Unlike obvious one-turn mistakes, this drift is subtle and accumulative – *the assistant's outputs become less aligned with the user's goal or the agent's role with each turn*. The prevailing reasons are **memory loss** (earlier info is no longer present or gets compressed in a way the model doesn't fully retain) and **compounding deviations** (small omissions or changes in tone each turn that add up) [19] [20].

- **Lack of Persistent State:** Many code assistants operate as "one-shot" or short-lived responders – they generate an answer and forget, relying entirely on the user's next prompt to provide state. This

linear *fire-and-forget pipeline* (`User prompt -> Code generation -> Done`) means the AI has no true memory of what just happened beyond what the conversation history repeats back to it [21]. If the conversation history isn't explicitly feeding back all relevant details, the agent effectively has **no long-term memory**. For instance, GitHub Copilot and others historically just use a sliding window of the file or chat, lacking any database of past interactions [10]. Without a mechanism to **persist the plan or past decisions** in memory, the model's internal state resets each turn, which can cause inconsistencies. It may recompute or reinterpret things differently over time, or drop tasks that were not re-mentioned. Newer agents like SWE-Agent or OpenDevon try to fix this by writing important info to a vector store or scratchpad that is re-injected when needed [10] [11] – but without such design, *an agent will inevitably have "memory blackouts"* on long tasks.

- **Summarization and Compression Artifacts:** To fit lengthy sessions into context, agents or frameworks often summarize earlier parts. If done naïvely, this can strip out nuance or emphasis that was important. For example, the initial instruction "always run tests after making code changes" might not survive a bad summary, or the summary might phrase it weakly. Each time context is trimmed or compressed, there's a risk of *losing critical constraints*. This leads to a form of *information decay* where the agent's understanding of the instructions becomes blurrier.

- **Goal Dilution and User Intervention:** In a long session, the user might ask follow-up questions or additional tasks that distract the agent from the original objective. If the agent isn't actively maintaining a notion of the top-level goal, it can get side-tracked. For example, while implementing Feature X, the user asks "What does function Y do?" – the agent might drop Feature X entirely and focus only on explaining Y from then on, unless it knows to come back to X. Without an *architectural self-awareness* or a contract reminding it of the overarching task, the agent's behavior becomes reactive (drifting into a Q&A mode) rather than agentic.

- **Model Priors and Style Reversion:** Large language models have strong default behaviors learned from training. If an initial instruction pushes it into a less common mode (e.g. very terse, highly structured outputs, or always commenting code verbosely), the model may slowly revert to its more "normal" style unless continuously reminded. This is partly due to entropy maximization – over many turns, slight deviations towards the default can accumulate. The *drift equilibrium* research suggests the process can stabilize (not infinitely worsen) but at a point where some of the original style/constraints are lost [22] [23]. In practice, we observe an assistant that was initially extremely detailed might gradually shorten its answers if not kept in check.

- **No System of Accountability:** If the agent does not have a way to **monitor its own adherence** to the instructions or any external checks, it's relying purely on the prompt (which, as noted, gets weaker over time). Human developers have practices like writing down requirements, using checklists, or running tests to stay on track. Absent analogous mechanisms, an AI agent can "go with the flow" of recent conversation and forget earlier requirements. For instance, the agent might begin by following a coding standard (because the system prompt said so), but after several user messages about something else, it forgets about the coding standard. There's no internal trigger that says "hey, that last output violated the style guideline you promised to follow."

- **Compounded Errors or Hacks:** If the agent made a mistake in an earlier step and it wasn't caught (or it was patched over ad-hoc), that can throw off later reasoning. Imagine the agent misunderstood the project architecture early on, and that misunderstanding isn't corrected – mid-

way through, its internal model of the system may be wrong, causing fragmentary or misguided actions. In essence, *early errors can snowball*. Without the ability to truly update its beliefs (beyond just what's in the text prompt), the agent might carry a flawed context that causes later behavior to seem random or shallow.

Why do these factors lead to shallow behavior? Because the agent falls back to surface-level operations once deeper guidance is gone. It may start coding without considering the broader plan (since it forgot it), or it might repeatedly ask the user for clarification that was already given (since that info scrolled out of context). In effect, the agent loses its "agency" and degrades into simpler pattern matching. It might still produce code, but it's no longer orchestrating a thoughtful multi-step solution – it's reacting turn by turn with decreasing awareness. This is precisely the scenario we want to prevent.

Enforcement Strategies to Maintain Agency

To counteract those failure modes, we can employ **practical enforcement mechanisms** that keep the agent on track throughout the session. The goal is to *persist the initial instructions and thoughtful behavior* so they don't fade, and to guard against the agent's natural tendency to drift or simplify the problem. Below we explore a range of strategies – from development environment features to prompt-engineering hacks – that help **enforce agentic behavior** continuously:

- **IDE-Integrated Orchestration and Logging:** One powerful approach is using an **agentic IDE (Integrated Development Environment)** that *bakes in structure and state* for the agent. Google's *Antigravity* is a prime example: it provides a dedicated "Agent Manager" interface where agents plan and execute tasks with full traceability [24] [25]. In such an IDE, every action the agent takes (like creating a file, running a command, editing code) can be logged as an **Artifact** – essentially a persistent record of the agent's plan, decisions, and results [26] [27]. This means the agent always has an external memory of **what it has done and why**, which it (and the human) can refer to. For example, Antigravity agents produce a **structured implementation plan** and a task list before coding [28]. The environment then ensures that at each step the agent knows which task it's on and can't just wander off – the plan is right there, and progress on tasks is tracked. Additionally, the *agentic IDE* paradigm often involves **multi-agent orchestration**: the ability to spawn sub-agents for sub-tasks and coordinate them [25] [24]. This enforces a kind of discipline, as the manager orchestrates who does what. From an enforcement perspective, these platforms provide **structure by design** – the agent isn't just free-form chat, it's operating within a framework (with UI sections for "Plan", "Tasks", "Results", etc.). This *wiring of the workflow* reduces the chance of drift; even if the model "forgets", the IDE will re-display the plan or remind it of the current task context. Moreover, traceability means any action can be audited or rolled back, deterring the agent from making random changes. In practice, an IDE-integrated agent might be prevented from, say, deleting a critical file without an explicit task in its plan to do so (the platform could prompt "This action is not in your plan, are you sure?"). By giving the agent a kind of **architectural scaffolding**, we constrain it to behave agentically.

- **Schema-Driven Prompts and Stateful Contracts:** A strategy at the prompt level is to use a **structured schema or "contract"** that persists through the conversation. Instead of relying on the initial instruction in free text, we embed the agent's mandate in a data format or an explicit checklist that gets echoed or updated every turn. For example, one might use a JSON or YAML snippet in the prompt that contains the agent's **Role, Goal, Constraints, and Done Criteria**. Each time the agent

responds, it must include this schema (possibly updated with progress). This acts like a **stateful contract** – the agent is continually presented with the written record of what it's supposed to do. Because it's part of the prompt, it won't be forgotten unless context is severely truncated. Modern LLM tooling even allows enforcing that the model's output conform to a given JSON schema [29] [30]. We can leverage that to require, say, an `"agenda"` field always be present. In practice, a schema-driven approach might look like: the system prompt provides a template like:

```
{
  "role": "Architect Assistant",
  "primary_task": "Implement feature X as per design Y",
  "subtasks_completed": ["Setup config"],
  "next_subtask": "Update database schema",
  "constraints": ["Follow coding standard ABC", "No external libraries"]
}
```

The agent then fills this out alongside its code output or explanation. By **injecting reminders and integrity constraints on every action**, we make it harder for the agent to go off-track – it would have to explicitly violate the structured fields that are in its face each time. Essentially, the schema serves as a persistent to-do list and rulebook. If the agent ever produces a schema missing a constraint or altering the goal improperly, our program can catch that (since it's machine-readable) and correct or reprompt the agent. This method has the benefit of not needing fancy external tools – it's enforced via prompt formatting and (optionally) a parser on the output. It's like giving the agent a contract at the start and having it sign and update that contract at each step.

- **Embedded Validators and Live Critiques:** *Wiring validators* into the agent's loop means the agent's outputs are immediately checked by an automated system, and feedback is provided or actions halted until issues are fixed. A classic example is integrating a **linter or test suite** that runs every time the agent writes code. Many agent frameworks now do this. For instance, an agent using *Mistral Vibe CLI* will apply code changes to a sandbox and run the relevant tests or linters automatically [31] [32]. If anything fails, the agent gets the failure output and is prompted to address it – all without human intervention. This creates a **self-correcting loop** that enforces technical correctness and adherence to specified norms. Linters can enforce style (if the agent's code style drifts, the linter flags it), type-checkers enforce architectural contracts (e.g. module A cannot import module B), and tests enforce that requirements are met. Essentially, these validators serve as **gates**: the agent's job isn't done (and in fact it is instructed to fix issues) until the checks pass. This ensures continuity because the agent can't just ignore an error and move on; it must integrate that feedback, which keeps it in an intentional problem-solving mode rather than outputting something and forgetting about it. Audit triggers are similar: we can set up **custom triggers** that audit the agent's actions for certain patterns. For example, if the project has a rule "no credentials in code", we can scan the agent's diff for any inserted secrets – if found, stop and prompt the agent to remove them. Another example: if we know the agent should not touch files outside a `src/` directory for this task, we flag any attempt to do so. These audits can be done via simple script or complex policy engine. The key is, they run *every step or at critical steps* to catch misbehavior immediately. This runtime guardrail concept is akin to unit tests for the agent's alignment: each action is validated against constraints before it's accepted [33]. If the agent deviates (say it tries to change a file not in its assigned scope), the guardrail intercepts it. The agent can then be forced to explain why it's doing that or revert the

change. By **automating oversight**, we reduce reliance on the agent's memory of rules – the environment itself enforces the rules consistently.

- **Token Signals and Prompt Anchors:** This is a more hacky prompt-engineering tactic: include special **tokens or phrasing cues** in the prompt that the model has been trained to associate with certain behaviors, thereby "steering" it. For instance, one could insert a phrase like: `"REMINDER: remain in agentic mode."` or a unique token sequence that was seen during fine-tuning to correspond to a certain style. If an agent was fine-tuned on data with system prompts containing "[AGENT_MODE_ON]", we could sprinkle that token periodically to jolt the model's weights into the right mode. Even without fine-tuning, repeating important instruction text verbatim in the prompt at strategic intervals can help counter drift. Research shows that *simple reminder interventions can reliably reduce divergence* from goals over long dialogues [23]. In practice, if we notice the assistant starting to produce shallow answers, we can inject a reminder like: "(*The assistant recalls its goal: to implement the feature following the plan and remaining verbose in its reasoning.*)" in the conversation. Because the model sees this text, it realigns its output distribution to match the described behavior. These token signals act like anchors, pulling the generation back to the desired path. One must be careful – overusing them can confuse the model or make outputs repetitive. But judiciously placed, they serve as mid-course corrections. Think of it as the equivalent of a teacher occasionally tapping a student on the shoulder to say "focus on the assignment." The nice thing is that, unlike the initial system message which might get diluted, these recurring reminders *re-inject the core directives* fresh into later context windows, keeping the model's attention on them [34] [35].

- **Standard Model Classifier as a Guardrail:** The **Collider** engine (which applies the Standard Model of Code analysis) can be used to enforce that the agent's changes remain *architecturally and semantically sound*. Collider can analyze code and label it across the 8 dimensions (D1–D8) and detect anomalies or anti-patterns. By running the agent's output through this classifier at each iteration, we create a feedback signal about design integrity. For example, Collider might classify a function before and after the agent's edit. If before it was `ROLE = Validator` and after the edit it's classified as `ROLE = Orchestrator` (just an example), we know the agent unintentionally changed the purpose of the function – a red flag. We could then prompt the agent: "Your edit seems to have changed the function's role/purpose from a simple Validator to something more complex. Is that intended?" Similarly, Collider's **lens outputs** can check if the *meaning* or intent changed: Lens R7 (Semantics) could summarize the intent of the code before vs after [14] [15], highlighting unintended shifts in logic. Moreover, the Standard Model defines certain *bad structures* (nicknamed **"antimatter"** when code is incoherent or overstuffed [36]). If the agent introduces a large blob of code that Collider flags as "antimatter" (e.g. a single function doing too many unrelated things, violating cohesion), we can automatically trigger a refactoring step or a warning. Essentially, we turn design principles into programmatic checks. A concrete implementation: after each code generation, run `collider analyze` to get the JSON metadata of the codebase [37] [38]. Examine fields like `role_confidence` or `layer` for each changed node. If a node's classification confidence drops or it switches layers (say a class was in "Application" layer and now appears as "Infrastructure" layer code), that might indicate the agent moved logic to the wrong place. The guardrail can then enforce a fix (like instruct the agent to relocate that logic or split the responsibility). Another use is detecting **cohesion or coupling violations**. The Standard Model classifier can identify when a single unit is playing "hybrid roles" (doing too many jobs) [39]. If an agent's edit causes that – e.g., a function now both writes to DB and sends network requests, making it a mix of a Repository and a Gateway – the system can catch it. In short, by integrating the *dimension/lens classifiers* as a watchdog, we ensure

the agent's solutions stay aligned with good architecture and the originally intended design. This is a novel form of guardrail that goes beyond syntax or tests – it's about *semantic alignment with design*. It effectively asks after each change: *"Is this code still serving the same role and purpose (D3, D7)? Does it obey the intended boundaries (D4) and state model (D5)? If not, correct it."*

- **Memory Injection and Reconciliation:** To maintain alignment across long sessions, sophisticated **memory management** is used. The idea is to *continuously feed the agent the important bits of past context,* even if the raw conversation is too long to include in full. One method is **selective retrieval**: store all past interactions (especially key decisions, plans, results) in an external memory (database or vector index), and for each new query, fetch the most relevant pieces to insert into the prompt. For coding agents, this could mean always reminding the agent of the *overall objective and previously completed sub-tasks*. For example, if sub-task 1 was done an hour ago, the agent might not recall it, but if we store "Sub-task 1: done (implemented X feature)" and retrieve it when the agent works on sub-task 3, it knows the context. Memory injection can also involve **periodic summaries** that are handcrafted to emphasize alignment. Perhaps every N turns, the system injects a summary: "Recap: We have done A, B, C. Our ultimate goal is Y. Next step is Z. Remember to follow constraint P." This acts like a strong alignment pulse that keeps the agent oriented. Research suggests that drift stabilizes and can be controlled to an extent [34] [23] – practical experience shows that regularly re-injecting the goal and key instructions prevents the worst of the drift. Additionally, toolchain design can enforce that *some prompts are always prepended*. For instance, some developers use an architecture where the **system prompt is not static** but is re-applied before every user query. So even at turn 50, the original role definition and rules are right there at the top of the model's input. This can be done in a custom CLI wrapper or agent loop. Essentially, the user's conversation so far might be truncated or summarized, but the system role instructions and any critical schema (as discussed earlier) are always included in full. This ensures that even if the model's own "memory" of earlier text fades, the instructions are literally present for it to condition on. Another part of alignment is maintaining a consistent **chain-of-thought** style. Agents often use approaches like ReAct (reason then act) or chain-of-thought prompting internally. By designing the prompt to always follow a pattern (like always output a plan before code), we enforce consistency. If the agent suddenly skips the reasoning step, the system can notice that deviation and ask for it (some frameworks have the agent output a "thought" and an "action" separately – if it doesn't, the framework can prompt "please provide your reasoning step"). This kind of *format enforcement* keeps the agent in a stable loop of thinking and doing, rather than spitting out fragmented answers.

In practice, robust agentic systems will combine multiple of these strategies. For example, an IDE like Antigravity provides the structured plan (schema + orchestration) and also runs tests (validators) and logs artifacts (traceability) [24] [26]. A terminal-based orchestrator like **LangChain or LangGraph** might not have a GUI, but it can implement similar enforcement: e.g. always use a JSON plan, always check outputs with custom validators, re-insert the system message every turn, etc. The overall theme is **redundancy** – we reinforce the agent's objectives and constraints through multiple channels (prompt text, external checks, environment design), such that even if one weakens, others catch it. This turns an otherwise free-form generative process into a **managed, stateful loop** where the agent is continually guided back to its path whenever it veers off.

Tool Comparisons: IDE-Integrated vs Terminal-Based Agents

Different tools and platforms support agentic behavior to varying degrees. Let's compare an **IDE-integrated agent** (like *Gemini 3 Pro in Google Antigravity*) with **terminal-based agents** (like OpenAI Codex CLI or Anthropic Claude CLI) in terms of how they maintain context and enforce aligned behavior:

[24] [25] *Table: IDE-Integrated vs. CLI-Based AI Agents – Features for Agentic Behavior Enforcement* [28] [26]

Aspect	**IDE-Integrated Agent (e.g. Antigravity + Gemini 3)**	**Terminal-Based Agent (e.g. Codex CLI, Claude CLI)**
Context & Memory	Extremely large context windows (Gemini 3 Pro offers ~1M tokens) reduce need for trimming [9]. Persistent state via the IDE: the agent's plan, task list, and prior actions are visually and programmatically stored. Agents can use semantic search on project files and history (the IDE provides project-awareness out-of-the-box).	More limited context (Codex: ~8k-32k tokens, Claude 2 up to 100k+ tokens). No built-in long-term memory – uses chat history only. Any persistence (like saving notes or using a vector store) must be handled by the user or an external orchestrator. Risk of losing earlier details unless manually re-provided.
Multi-step Orchestration	Native support for multi-step workflows: the IDE can spawn sub-agents, schedule background tasks, and sequence tool usage autonomously [25]. The "Agent Manager" coordinates steps and keeps the agent(s) on the predefined workflow. Agent actions (code edit, test run, browser open) are part of a managed plan.	Typically single-agent loop unless the user scripts something. The agent responds to one command at a time. No inherent concept of a "plan" or multiple agents – though the user could manually prompt it to make a plan. Sequencing of actions relies on the user's prompts or a custom harness (e.g. writing a Python script to call the agent, then a tool, then agent, etc.).
Interface & Reminders	Rich UI: The agent's plan and goals can be persistently shown in a sidebar, acting as a constant reminder. Artifacts like "Implementation Plan" are always visible [28]. The agent also sees these in its own input. The IDE often injects the current task context automatically into the prompt (e.g. "You are working on Task 2: update the API endpoint..."). This reduces forgetting – the context isn't just in the agent's "mind", it's in the environment.	Plain interface (terminal or chat): The agent sees only the text we programmatically feed it. Any reminders or plan must be included in the prompt by the user or wrapper. There's no persistent visual context; if the user doesn't explicitly remind the agent ("Now we are doing X..."), it may forget. Some CLI tools support a system prompt, but it remains static unless managed.

Aspect	**IDE-Integrated Agent (e.g. Antigravity + Gemini 3)**	**Terminal-Based Agent (e.g. Codex CLI, Claude CLI)**
Automatic Validation	Strong integration with tooling: The IDE can auto-run tests, linters, type checkers after the agent's code changes, then feed results back to the agent. For example, Antigravity agents verify their own output and produce artifacts like test results [25] [27]. This tight feedback loop enforces quality without user intervention.	Basic integration: The CLI agent might output code to the terminal, but running it or testing it is up to the user. Some CLI implementations (Claude CLI or others) might have simple commands like "run" or use a plugin for executing code, but it's not as seamless. Without an external orchestrator, the agent won't know to re-run tests unless explicitly told each time.
Traceability & History	Every action is logged and can be reviewed. The agent cannot "hide" an operation – if it deletes a file, that is recorded in the artifacts or version control. Developers can audit what happened and why [24] [26]. This history can also be leveraged by the agent (the IDE might automatically remind the agent of past mistakes or highlights from the log). Essentially, a **shared memory between human and agent** exists.	History is just the chat transcript. If something was done 20 turns ago, it's buried unless the user scrolls up or the agent was made to summarize it. There's little notion of *why* the agent did something except what it said at that time. No structured log of actions unless user manually keeps one. This makes it harder to audit; also the agent itself has only the chat to rely on for history of its actions. If not carefully prompted, it might not recall why it made a past decision.
User Interventions	More opportunities for high-level guidance: In an IDE manager view, a human can tweak the plan or rearrange tasks if needed, and the agent will adapt to that updated plan. The interface separates planning from execution, so a user can step in at the plan level. This prevents the agent from going too far down a wrong path – a kind of supervised agency.	All interactions are via the same chat. While the user can of course correct the agent or give new instructions, it's in-band with everything else. There is no separate "planning console" to adjust without confusing the agent. The user must carefully prompt to course-correct, which the agent might or might not perfectly incorporate. It's harder to manage at a meta-level without an orchestration UI.

Overall, **IDE-integrated agents offer a more structured, monitored environment** which naturally promotes agentic behavior. They reduce the cognitive load on the agent (less info to recall unaided) and on the user (less need to babysit each step), by automating context management and enforcement. In contrast, **terminal-based agents are more free-form**, which gives flexibility but means one must explicitly apply many of the enforcement strategies we discussed – usually via an external orchestrator script – to achieve similar reliability. For example, developers using a CLI agent often end up writing a driver code that does: *prompt agent for plan -> feed plan back when needed -> after code gen, run tests -> if fail, show error to agent -> etc.*. That's essentially custom-building what an agentic IDE provides out-of-the-box.

It's worth noting that as of 2025, the gap is closing: CLI agents are getting better tooling (e.g. Anthropic's **Claude CLI** can connect to a filesystem and follow a multi-turn tool-using chain [40], and frameworks like LangChain offer guardrails libraries). Meanwhile, IDEs like Antigravity demonstrate what *fully integrated* agent loops can do – e.g., maintain *100+K tokens of project code in context, and orchestrate web browsing and terminal commands simultaneously*. The Collider + Standard Model ecosystem can be seen as complementary here: Collider provides the deep analysis and classification that an IDE agent can use for insight, while the Standard Model gives a language to talk about architecture in the prompt ("This function is a Repository in the Domain layer..."). An IDE agent could automatically annotate code with Standard Model metadata and thus *be aware of dimensions like D3 Role and D7 usage context as it works*, something a basic CLI agent wouldn't know unless explicitly told.

Recommended Architecture for Agentic Resilience

Bringing these ideas together, we can outline an **architecture for AI code assistants that are resiliently agentic** – meaning they *stay on course* and retain high-level autonomy even over long, complex sessions. The architecture involves several layers working in concert:

1. **Dual Memory System (Short-term + Long-term):** The agent is equipped with a short-term memory (the immediate conversation window or scratchpad) and a long-term memory (persistent storage of important info). Designate a **Memory Manager** module that stores key data: the project state, plans, decisions, and important conversational points. This could be a vector database or a structured knowledge base. The agent's core prompts at each step should be augmented by relevant retrievals from this memory (e.g. summary of the plan, or the content of a file it decided to edit, etc.). This ensures continuity. As literature suggests, adding persistent memory helps agents track state across tasks [10] [11], at the cost of some complexity. A robust design might use hierarchical memory: e.g. summary of last turn (short-term cache), summary of the overall session (mid-term), and full database of specifics (long-term) [41] [42]. The Memory Manager orchestrates what to fetch and inject to keep the context alive.

2. **Structured Plan and State Representation:** At the heart is a **Planning Component** that uses a schema or state machine to represent the agent's task. This could be an explicit *state object* containing the current objective, sub-task list, and progress (like the JSON contract mentioned earlier). The agent should update this state as it works (or have the system update it). By having a single source of truth for "what are we doing," both the agent and any outside monitors are synchronized. A *stateful agent loop* might look like: `state = plan(task); while(!done){ step = agent.act(state); result = environment.execute(step); agent.observe(result); update state; }`. The plan/state is threaded through each iteration. This prevents the flakiness of just letting the agent generate whatever – it must always consider the state and ideally output a new state or action that's consistent. In essence, **make the agent operate like a deterministic state machine augmented with creativity**: the structure (states, transitions) is fixed, but the content of actions is generated. LangGraph's approach of treating the workflow as a graph with nodes and edges that loop on errors is a good blueprint [43] [44]. The agent's "brain" should be this orchestrator that can handle loops (retry on failure), conditionals (if error type X then do Y), and concurrent sub-tasks. By explicitly encoding these, we take the burden off the agent to implicitly remember to do things like test after coding – it's in the loop by design.

3. **Validation and Guardrail Layer:** Surround the agent with a **gauntlet of validators**. Each output or action from the agent goes through filters:

4. **Syntax/Schema Validator:** Does the output conform to the expected format (e.g. JSON schema, or at least valid code syntax if code)? If not, reject it and ask the agent to fix the format. This catches cases where the agent might drift into explanatory prose when you expected JSON, or otherwise break the protocol.
5. **Policy/Safety Guardrails:** If there are forbidden operations (like accessing the internet in certain contexts, or modifying certain files), a guardrail should catch them [33]. Many frameworks provide rule-based or model-based classifiers for this. For instance, OpenAI's guardrails or Guardrail SDK can intercept and refuse outputs that violate compliance or user-defined rules.
6. **Semantic/Design Validator:** This is where the Standard Model's classifier can be used. After the agent proposes a code change, run an analysis to see if it introduces any known anti-pattern (e.g. a dependency cycle between layers, or a function too large). If yes, either automatically refactor or prompt the agent: "Your solution seems to violate the layering constraint. Please adjust to preserve proper layering." (For example, if an agent writes UI code that directly calls the database, the Standard Model D2/D4 dimensions would reveal a layer violation – the fix might be to insert a service layer in between.)
7. **Tests & Runtime Validator:** Have the agent's output tested in a sandbox. This includes running unit tests, integration tests, or just executing the code to see if it throws errors. The results should funnel back to the agent's input on the next iteration [45] [31]. If a test fails, the agent knows it must debug before proceeding. This loop continues until tests pass or a limit is reached.
8. **Performance/Quality Gates:** For more advanced setups, one could also enforce performance benchmarks (e.g. if the agent writes a solution that is too slow, a test could flag that) or style guides (lint rules as mentioned). These ensure the agent's output isn't just functionally correct but also meets quality standards.

All these validators essentially act as **automated critics/mentors** alongside the agent. They keep the agent "honest" and on spec. Notably, the validators themselves can be powered by simpler algorithms or even LLMs specialized in critique. For example, one might use a smaller model fine-tuned to detect if the code deviated from the spec or not. This is analogous to how one might have a second model watch the first (an idea in some academic work where a 'guardrail model' evaluates the 'generative model' [46]). However implemented, the architecture should allow plugging in these checks easily.

1. **Interactive Editor & Version Control:** The agent should operate on a **working copy** of the code (in-memory or a branched git repository) so that all changes are tracked. By using version control, every change is a commit that can have a message explaining it. The agent can be required to produce commit messages which forces it to articulate its reasoning for the change (this in itself is a form of reflection). If something goes wrong, the system can roll back to a previous commit and either try again or take a different approach, thus implementing a form of backtracking. This aligns with the idea of *program state tracing and replay* for agents [47]. An agent can "checkpoint" states and revert if needed, rather than getting stuck in a faulty state.

2. **Dimension/Lens Contextualization:** To imbue architectural awareness, the agent architecture can incorporate **Standard Model metadata as context**. For instance, before the agent modifies a function, the system can feed the agent a snippet of metadata: "Note: This function is classified as Role=Cache (stores data temporarily) in the Infrastructure layer, and has Effect=Write (it writes to storage) [48]." By explicitly providing these labels (which can be obtained via static analysis like

Collider), the agent is less likely to do something contradictory (like altering it to do computation instead of caching). Essentially, treat the Standard Model info as part of the agent's observation: just like it sees code text, it also sees the "code semantics." Over time, the agent could even be fine-tuned to ingest this kind of metadata natively, making it inherently architecture-aware. In our architecture, this would mean whenever the agent loads a file or piece of code to work on, it also gets the associated dimension vector (D1–D8) and maybe lens insights. This provides a **guardrail from the inside** – the agent's own prompting contains cues about what things are (e.g. "This is a *Controller* class – it mediates between UI and core [49] [50] ."). If it knows that, it is more likely to preserve that role.

3. **Human Oversight Interface:** While the goal is autonomy, a resilient architecture should have a channel for human intervention when needed – a sort of "steering wheel" that a developer can grab if the agent veers off. This could be as simple as a dashboard showing the agent's current plan, actions, and any warnings from validators, with the ability to pause or modify the plan. In an enterprise setting, one might require the agent to get human approval when touching very sensitive parts of the codebase. This ensures that if all automated checks fail to catch a misalignment, a human can still step in by reviewing the agent's artifacts (commit diffs, plan documents, test results) and then either re-align the agent ("Actually, don't do X, do Y") or handle that piece manually.

To illustrate, consider how these components would work in a scenario: The user gives a high-level feature request. The agent (via Planning Component) generates a multi-step plan and a JSON state documenting sub-tasks. This is shown to the user (via the oversight UI) and also stored. The agent starts with sub-task 1, retrieves relevant context (files to edit, their Standard Model metadata, etc.) from memory. It writes code for sub-task 1. The output is caught by the validators – syntax OK, passes style linter, but one unit test fails. The Test Validator feeds the failure back. The agent enters a refine loop, debugs (maybe using the error message). Once it passes, the change is committed. The state is updated: sub-task 1 done. The agent moves to sub-task 2, and so on. At each step, the contract in the JSON (state) is reiterated to the agent, so it knows what's done and what's next. Midway, if a design issue is flagged (say the Semantic Validator notes that the code complexity is growing too high in one function, violating a complexity threshold), the agent might create a new sub-task on the fly to refactor that function (or the orchestrator does). Eventually, feature is implemented, all tests pass, and the state says "all subtasks done." The agent might then summarize the changes and open a pull request artifact as final output.

Crucially, this architecture ensures **the agent cannot easily escape the loop or ignore the plan** – the structure is reinforced by both the program logic and the prompt content at each iteration. Even if the underlying model has a tendency to wander, the system around it corrals it back in line.

Implementation Recipes

To implement the above strategies in practice, here are some concrete "recipes" and tips:

- **Persistent System Prompt:** Make your system or initial prompt *reentrant*. That is, design your code so that before each model call, you prepend the core role and rules. Don't rely on the model to recall them from earlier turns. For example, if using OpenAI's API, always include something like: `system_message = "You are an AI agent following this plan... [plan JSON] ... Here are the rules: ..."` followed by the conversation. This way, no matter how long the

conversation, the model always sees the important instructions. This simple trick addresses a huge part of mid-session drift and is straightforward to do.

- **Vector Memory for Q&A and Codebase:** Integrate a vector search on your code repository and past conversation. When the agent asks or needs information (e.g. "Where is the config parsed?"), your wrapper can automatically do a vector similarity search on the codebase or documentation and feed the results. Likewise, keep a vector index of dialogue chunks (or use embeddings of the plan and decisions) so you can retrieve those if relevant ("reminder: earlier you decided not to use library X."). There are libraries and LangChain modules that facilitate adding such retrieval steps. The key is to trigger these retrievals at appropriate times (like before agent starts a new subtask, or when it queries something). This gives the agent richer, *context-aware information* without requiring it to explicitly ask (which it might forget to do).

- **Use Function Calling or Tools API:** Many LLM platforms now support *function calling*, where the model can output a JSON blob calling a tool, and you as the developer get that and execute the tool, then return the result. Use this to your advantage for enforcement: define "tools" for running tests, running linters, analyzing code with Collider, etc. The agent can then be prompted in a ReAct style to use these tools whenever appropriate ("If you want to run tests, call the `run_tests` function."). More proactively, after the agent outputs code, you can force a test run tool call before finalizing the agent's answer. By structuring the interaction as tool usage, you make these validation steps first-class citizens in the conversation. The agent might even learn to predict errors and run tests pre-emptively. For instance, it could call a `analyze_design` function that returns Standard Model analysis of its code, then see the results and decide to adjust. This is a more advanced pattern, but it's achievable with function-calling models. Essentially, *turn guardrails into tools* that the agent can invoke or that the orchestrator invokes between agent turns.

- **Adopt an Agent Framework:** Don't reinvent the wheel if not needed – frameworks like **LangChain**, **OpenAI Functions**, **Microsoft Guidance**, **Haystack**, etc., provide scaffolding for chaining prompts, tools, and memory. For example, LangChain has a concept of Chains and Agents that can automatically loop on a task until a condition is met, as well as integration with the **Guardrails library (GuardrailsAI)** to validate outputs. By using these, you can declaratively set up (for instance) a JSON schema that the output must follow, or a condition that if a certain word appears the agent should stop. These save time and encode community best practices.

- **Monitor Token Usage & Truncation:** Keep an eye on how close you are to context limits. A recipe here is to implement an **adaptive context strategy**: if the conversation is short, include full details; if it grows long, start summarizing or dropping less-important parts, but always keep the critical ones (like the plan and constraints). You can maintain a priority list of context elements. Some developers use techniques like *LIFO drop-out* (drop oldest turns first, since they're presumably least relevant now – but not the system prompt!) or train a separate summarizer model to compress older interactions into a blurb that retains important info. Implementing a **"context equilibrium"** as researched by Dongre et al. can be an aspirational goal – they showed drift can be bounded with interventions [35]. In practice, an intervention could be as simple as "if the assistant's last 3 outputs show deviation, explicitly reinsert the entire initial prompt next turn."

- **Use Git Hooks or CI for an External Check:** If your agent is committing code, you can leverage existing DevOps tools as enforcement. For example, set up a git pre-commit hook that runs a linter/

test, and if it fails, reject the commit (and thus signal the agent it must fix something). Or use a continuous integration (CI) pipeline that the agent triggers after a batch of changes; if CI fails, that feedback goes to the agent. This way you piggyback on the robust infrastructure already in place for human code quality – treat the agent as just another developer whose code must pass CI.

- **Teach the Agent with Examples:** You can prompt the agent with demonstrations of maintaining context. For instance: *"Example: [conversation where an agent consistently refers back to the plan]. Now do likewise."* Few-shot prompting can reinforce behavior. If the model sees an example agent being diligent (e.g. "Plan step 2 of 5 complete. Moving to step 3..."), it may mirror that. This acts like training wheels – giving a pattern to follow. It's not bulletproof, but it nudges the style.

- **Logging and Analytics:** Enable detailed logging of the agent's behavior and outcomes of validations. Analyze these logs to find where things went wrong when they do. You might discover, for example, that after ~15 turns your reminder frequency was too low, or that a particular regex in your guardrail caused false positives that confused the agent. Using these insights, refine your enforcement mechanisms (maybe increase reminder frequency, or relax an overzealous guard). Essentially, treat this system as you would any complex software – monitor and iteratively improve it. Over time, you'll identify the weak points where the agent still goes off-track and can add a new guard or tweak prompts accordingly.

Implementing all of this is non-trivial – it's like building an **autopilot for coding**. But even a subset of these recipes can dramatically improve an agent's reliability. For instance, a simpler implementation might be: always prepend a short reminder of goals, use a vector store for file lookup, have a loop that runs tests after generation, and if failure, have the agent fix it. That covers continuity, architectural context (because the agent can find relevant files), and error correction – yielding a far more agentic behavior than a naive approach. As one Medium article put it, the shift from "one-shot" tools to self-healing, continuous agents is the paradigm change needed for true coding co-pilots [51] [52] .

Limitations

While these strategies greatly enhance an AI agent's autonomy and alignment, **there are limitations and challenges** to acknowledge:

- **Complexity and Overhead:** Layering many enforcement mechanisms makes the system complex and potentially slow. Each validator or memory operation is additional computation (e.g., running a Collider analysis on every code change or doing multiple vector DB lookups). This can increase latency and cost. There's a trade-off between strict enforcement and efficiency. In a real IDE, too much delay between agent actions can hurt user experience. Also, more components mean more potential points of failure (the agent might get stuck in a loop if a guardrail mis-fires, etc.). Engineering the orchestration requires careful thought to avoid infinite loops or contradictory signals to the agent.

- **Model Limitations:** The underlying LLM might still be prone to mistakes or might not respond to our interventions perfectly. Not all models are equally amenable to following schemas or tool feedback. For example, if using an older Codex model, it might ignore JSON format instructions and output text anyway. Some guardrails rely on the model's compliance – if it decides to ignore the format and outputs a huge unwieldy text, your parser might break or the agent state might

desynchronize. Newer models (GPT-4, Gemini, Claude 2, etc.) are better at this, but it's not foolproof. Moreover, if the model misinterprets a reminder or a piece of memory (summaries can introduce subtle errors), it could lead the agent astray confidently.

- **Incomplete or Inaccurate Classification:** If using automated design checks (Standard Model classification, etc.), note that these classifiers have their own error rates. The code semantic analysis might misclassify something (maybe a function is borderline between two roles, and it chooses the wrong one). Guardrails based on that could then mistakenly flag correct agent behavior as incorrect, forcing needless changes or confusing the agent. For instance, the agent might write a function that is fine, but the classifier yells "hybrid role" due to a quirk – the agent then tries a convoluted refactor to satisfy it, potentially making the code worse. Thus, automated guardrails need tuning and sometimes human confirmation. We wouldn't want to replace one problem (drift) with another (overzealous micromanagement by flawed validators).

- **Loss of Creativity/Flexibility:** Enforcing a strict plan and schema can sometimes backfire if the situation changes. What if mid-task, the user or environment introduces a new requirement? A highly rigid agent might struggle to deviate from the initial contract. For example, if the plan turns out to be wrong, a very strict enforcement might keep the agent marching towards a flawed goal. Ideally, the agent should be able to recognize when to revise the plan. Some architectures include meta-reasoning to allow plan adjustments, but it's tricky – you must allow the agent to modify the plan *without* it being seen as "drift." Similarly, too much validation might nip creative solutions in the bud. Perhaps the agent has a novel approach that breaks a usual pattern (hence triggers a guard), but is actually valid. The system might force it back to a conventional approach. In essence, strong guardrails can constrain the solution space and the agent may miss some clever or efficient implementations because they look "unusual."

- **Scaling to Very Large Projects:** Memory and context techniques can falter if the project is enormous (millions of lines, or highly complex interdependencies). Even 1M token context might not fit an entire enterprise codebase. Vector search helps, but if the agent needs a holistic understanding (like a sweeping refactor across 100 microservices), no current memory strategy completely solves that. The agent might still have to tackle such tasks piecewise, and continuity over weeks of work (beyond a single session) is an open challenge. We might need persistent agent state that lives beyond a single chat session – that introduces questions of how to store and re-load it reliably (the JSON contract could be saved to disk between runs, for instance). Ensuring the agent's "personality" or coding style stays consistent over long periods (days/weeks) when it might be restarted or updated is also hard.

- **User Compliance and Understanding:** These systems can be complicated to operate. A developer might need to understand the schema and orchestrator to effectively steer the agent. If the user doesn't know about the internal plan representation, they might inadvertently give an instruction that conflicts with it (e.g., telling the agent "skip step 3" in plain English might confuse it vs. using the proper interface to remove step 3 from the plan). There's a UX challenge to present this sophisticated agent in a way that's intuitive. Antigravity's approach of a separate Agent Manager UI is one solution, but it might not fit all workflows. Some users just want to chat in natural language entirely – balancing that with the structured approach is tough.

- **Edge Cases and Adaptability:** Agents can encounter novel situations the designers didn't anticipate. No matter how many guardrails we add, something can slip through or cause weird behavior. For example, a validator might handle known test failures but what if the code doesn't compile at all? The agent might get stuck if it never gets to run tests. Or an external API the agent calls might hang – does the agent know to time out? These kinds of error-handling and off-nominal scenarios need to be thought out. Traditional software can enumerate those, but an AI agent can do unexpected things. It might output a huge blob of text that crashes the JSON parser, or it might decide to solve a task in an extremely roundabout way that breaks our loop logic. Robustness requires a lot of testing and iterating.

- **Integration with Human Workflow:** In team settings, an AI agent must play nicely with human developers. If an agent commits directly to a repository, there might be cultural or process issues (e.g. code review: do humans review the agent's commits? If so, the agent's speed might be slowed waiting on that). If the enforcement is too rigid, developers might turn it off ("this thing blocks me too often with false alarms"). So adoption requires tuning the level of strictness to what people are comfortable with, and proving the value (if it catches real issues and saves time, people tolerate it; if it nags wrongly, they won't).

In summary, while our enforcement strategies significantly mitigate the common failure modes of agentic systems, **they don't guarantee perfection**. We must be mindful that we're essentially building *complex socio-technical systems* (AI + tools + humans). There will be an ongoing need for oversight, tweaking, and possibly relaxing some constraints at times. However, even with limitations, these strategies raise the floor: an agent with them is far less likely to completely lose the plot mid-session, and failures when they occur will likely be more benign (e.g. a test failing which we catch, rather than silently introducing a bug or forgetting a requirement).

Future Directions

The field of AI coding agents is rapidly evolving. Looking ahead, several developments could further **enhance agentic behavior and its enforcement**:

- **Intrinsic Model Improvements:** Future LLMs (beyond GPT-4, Claude, Gemini, etc.) might come with architectures explicitly designed for long conversations and task persistence. Research like *context equilibrium* [20] [19] and *long-context training* is pointing toward models that inherently resist drift. Imagine a model trained on 100-turn dialogues where it was rewarded for consistency with initial instructions – such a model would need less external prodding. If "agentic fine-tuning" becomes a thing, models could internalize patterns like always making a plan, self-checking code, etc., from the get-go. OpenAI's function calling and similar features are early steps; we might see more granular control in prompting, like being able to pin certain tokens/vectors in the model's context eternally (so the notion of "goal = X" never fades). Additionally, improvements in **transformer architectures for long context** (like efficient attention or segment memory) will let models carry more information without forgetting, reducing the need for our manual memory injections.

- **Neurosymbolic and Hybrid Approaches:** We might combine symbolic reasoning with neural flexibility more deeply. For example, an agent could learn to translate a natural language spec into a **formal schema or automaton** that it then follows rigorously. This bridges the gap between human instructions and machine execution. Concepts from software engineering (statecharts, Petri nets for

workflows, etc.) could be integrated. A lens to consider is: the Standard Model of Code or any schema we use could itself be part of the model's reasoning – e.g., the model might prompt itself with "Check design: [Collider output]. If design issues, fix them." One can imagine *embedding the Standard Model classifier within the agent's neural network via multi-task learning*, so it would implicitly know the dimensions of any code it writes. That could pre-empt many alignment issues (the agent would "feel" that it's e.g. mixing roles wrongly and adjust before even outputting).

- **Reinforcement Learning & Self-Correction:** Going beyond supervised fine-tuning, using reinforcement learning (RL) to reward long-horizon coherence is promising. For instance, an agent could be trained with a reward that measures how well it stuck to a given instruction set over 50 turns. Or reward it for not needing human correction. *RL with human feedback (RLHF)* could be used not just for making assistants polite, but for making them follow through on multi-step tasks. An exciting prospect is agents that **simulate a pair programming scenario** – where one agent plays the "navigator" (high-level planning, reviewing) and another is the "driver" (writing code), and they provide feedback to each other. This dual-agent setup could be RL-trained so that the navigator catches drift in the driver and vice versa, and only reward when both are aligned in the end (some research already hints at using multiple models to supervise each other).

- **Personalized and Contextual Behavior Models:** In the future, agents might build a *profile of the project and team* they are working with. They could dynamically adjust how strict to be or how to behave based on context. For example, if the project is a safety-critical system, the agent might enter a "high-assurance mode" with extra validation and very formal output. If it's a quick script in a hackathon, the agent might be more flexible and creative. This kind of adaptive alignment – knowing when to rigidly enforce every rule and when to bend a little – could be mediated by higher-level AI that classifies the nature of the task or user preferences. Dimension classifiers might expand beyond code to **situation dimensions** (D9 Intent, as noted in the Synthesis Gap doc [53] [54], or perhaps a dimension for criticality). The agent could then self-regulate: e.g., if `Intent = Ambiguous`, maybe ask the user for clarification (ensuring alignment before proceeding), etc.

- **Better Human–Agent Collaboration Interfaces:** We will likely see improved UIs and experiences for working with agentic AI. Antigravity is one vision; others might integrate into IDEs like VS Code or JetBrains such that the agent's state is visually represented (perhaps as annotations in the code or as a dashboard). **Visual diagramming of the agent's plan** could make it easier for humans to spot if it's going off-course. Additionally, features like "timeline scrubbing" could let you rewind the agent's state to any previous turn (since everything is logged) and fork a different approach – effectively treating agent sessions like Git branches that you can diff and merge. All this would make maintaining long-running agent sessions more manageable and auditable. We might also get standardized formats for agent plans (somewhat like how UML diagrams standardize design, we could have an "Agent Workflow Markup Language" that any tool can read and present nicely).

- **Cross-Agent Communication and Swarm Intelligence:** Today, we mostly consider one agent working solo (or maybe a couple in orchestrator/worker roles). In the future, one could deploy a *team of specialized agents* – e.g., one agent focuses on testing, one on security review, one on coding – essentially an AI development team. They would communicate among themselves (in a controlled way) to verify each other's work. For instance, a "Reviewer" agent could critique the "Coder" agent's output (similar to validators but in natural language) and the Coder would improve it. This is like having multiple expert systems, each with agentic behavior in their domain, collaborating. Such

setups could maintain alignment because each agent keeps the other in check (the security agent will keep reminding about security requirements, etc.). However, orchestrating multi-agent dialogues adds complexity – it would need careful conversation management to avoid them drifting collectively.

- **Continuous Learning from Deployment:** In the future, agents might learn from their mistakes across sessions. Imagine an agent that, after each session, logs what went wrong (e.g., "I drifted and had to be corrected about X"). These could be fed into an offline training process to make the next version less likely to drift in that way. Essentially **online improvement** – the more the agent is used, the better it gets at staying on task (assuming a mechanism to retrain or fine-tune on collected data). This raises safety concerns, but also potential: the Standard Model's structured data could be used to measure improvements ("last month the agent had 5 cohesion violations flagged by Collider, this month only 1 on similar tasks, good.").

- **Expanded Standard Model and Collider Capabilities:** The Standard Model of Code itself might evolve (we saw hints of adding D9 Intent, D10 Language, etc. [53] [54]). If Collider can capture even more about code (like intent or documentation alignment), it can give stronger signals to agents. Also, Collider could move from passive analysis to **prescriptive suggestions**. Instead of just flagging "this is antimatter," it might say "this class is doing too much – consider applying the Single Responsibility Principle by splitting it." Feeding such suggestions into the agent could help it learn better design autonomously. The agent+Collider combo could essentially become a self-improving loop: agent writes code, Collider scores design, agent adjusts, and so on, approaching an optimal solution by design metrics.

- **Generalization to Other Domains:** Today's enforcement discussion is about code agents, but similar principles will apply to agents writing documents, executing business processes, etc. The notion of maintaining purpose (like an AI writing a long report staying on theme) has parallels. Techniques like schema enforcement and reminders are already used in long document generation and customer service bots. Advances in one domain will cross-pollinate. For example, a discovery in how to prevent topic drift in chatbots could be applied to preventing task drift in coding agents. Collaboration between different AI fields (NLP long-form generation, dialog systems, robotics planning) will likely yield new guardrail techniques that we can borrow for coding agents.

In conclusion, the trajectory is towards **AI agents that are reliable, transparent partners** – and achieving that requires both smarter AI and smart infrastructure around the AI. The Collider + Standard Model ecosystem is an excellent foundation: it provides a language to describe what code is and should be (dimensions, lenses, design principles). Building on that, we enforce that our AI coding assistants constantly respect those descriptions – effectively *aligning the agent's local actions with the global architecture* of the codebase. By combining architectural insight with modern AI orchestration, we move closer to agents that truly understand what they're building and *why*, not just how to satisfy a prompt in the moment. Each improvement in maintaining agency is a step from "code generator" to **autonomous software engineer** – one who can be trusted with complex tasks without losing focus or integrity mid-way. The future will no doubt bring us even more creative solutions to keep these digital "engineers" on track, and it's an exciting space where software engineering principles and AI research intersect deeply.

Sources:

1. Dongre et al., *"Drift No More? Context Equilibria in Multi-Turn LLM Interactions,"* 2025 – Defining context drift as a slow erosion of intent over a conversation [55] [18] and showing that periodic goal reminders can reduce divergence [22] [23].

2. **Standard Model of Code Documentation** (Collider Project) – Definitions of the 8 Dimensions (D1–D8) and their role in code semantics [48] [56], and the 8 Lenses including Semantics (purpose/intent) [14] [15]. Illustrative analysis of code issues using Standard Model classifications: e.g. detecting a cohesion violation when a function mixes roles (flagged as "Hybrid Role – Antimatter") [39].

3. Google Antigravity – *Agentic IDE* announcement and Tessl technical review – Describing Antigravity's agent-first features: dedicated Agent Manager, structured plans and artifacts, multi-surface orchestration (editor, terminal, browser) [24] [28], and traceability for auditing agent actions [24] [26].

4. Muhammad Awais, *"Your Code Assistants Are Stuck in the Past. Here's How 'Self-Healing' AI Will Actually Write Software,"* Medium, Dec 2025 – Discussing the shift from one-shot code generation to looped, self-correcting agents with a stateful brain (LangGraph) and hands (Vibe CLI) [51] [43]. Emphasizes the importance of feedback loops (running code, capturing errors) to unlock true agentic behavior [21] [31].

5. Liu et al., *"AI Agentic Programming: A Survey,"* 2025 – Overview of agentic coding systems. Notes how tools like Copilot lack persistent memory (using only sliding window) versus newer agents using vector stores for memory [10]. Stresses the need for structured memory and context tracking to avoid repeating mistakes or forgetting earlier steps [16] [57]. Also highlights that current LLMs, even with long contexts, often need to leverage program structure for focus [58].

6. LangChain Documentation – *Guardrails* for LLMs: validating and filtering outputs at key points to ensure compliance [33]. Illustrates the principle of inserting checks in an agent's execution loop to intercept undesired content or format.

7. **Collider Storage Architecture** – Details on how Collider outputs a JSON with fields for each code node including dimensions and lenses [37] [38]. This structured data enables programmatic checks on code properties (e.g. role, layer, side effects) after agent modifications.

8. Standard Model *Foundational Theories* – Inspiration for multi-dimensional analysis. E.g., Koestler's holons (whole-part relationships) integrated as containment hierarchy and Role/Boundary dimensions [7]. Halliday's linguistic metafunctions mapped to code dimensions (Interpersonal ~ Role/Boundary, Textual ~ Layer/Lifetime, etc.) [59] [49], reinforcing why considering multiple facets (purpose, layer, state, etc.) is crucial to understanding and enforcing context in software agents.

9. Dong et al., *"Drift in LLM-based Agents"*, Adobe Research blog – (Hypothetical reference summarizing practical drift issues and mitigation, aligning with Dongre et al. 2025).

10. OpenAI Function Calling & Tools – (Docs and examples) showing how forcing structured function outputs can enforce output schema and enable agents to act in loops (e.g. tool->observation->tool).

1 2 3 8 9 10 11 16 40 41 42 47 57 58 AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities
https://arxiv.org/html/2508.11126v1

4 5 12 13 14 15 36 39 48 49 50 56 59 THEORY.md
file://file-7aTzMHSPLhbjJa8Kx9ZyWD

6 7 FOUNDATIONAL_THEORIES.md
file://file-DffAKLYGkg6uNfLqU5Yumy

17 18 19 20 22 23 34 35 55 Drift No More? Context Equilibria in Multi-Turn LLM Interactions
https://arxiv.org/html/2510.07777v1

21 31 32 43 44 45 51 52 Your Code Assistants Are Stuck in the Past. Here’s How ‘Self-Healing’ AI Will Actually Write Software. | by Muhammad Awais | Dec, 2025 | Medium
https://blog.devwithawais.com/your-code-assistants-are-stuck-in-the-past-3b3a3f867653?gi=a5df53f0b4c5

24 25 26 27 28 Antigravity: Google’s next step in agentic development
https://tessl.io/blog/gemini-3-meets-antigravity-googles-next-step-in-agentic-development/

29 How to get structured output from LLM's - A practical guide
https://builder.aws.com/content/2wzRXcEcE7u3LfukKwiYIf75Rpw/how-to-get-structured-output-from-llms-a-practical-guide

30 Structured Output Support for Prompt Experiments - Langfuse
https://langfuse.com/changelog/2025-09-30-structured-output-experiments

33 Guardrails - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/guardrails

37 38 STORAGE_ARCHITECTURE.md
file://file-158oWtaycaFNCotChdH7vi

46 An Introduction to “Guardrail” Classifier-Trained LLMs - AightBits
https://aightbits.com/2025/11/09/an-introduction-to-guardrail-classifier-trained-llms/

53 54 SYNTHESIS_GAP_IMPLEMENTATION.md
file://file-7kCfFwGdpxnnhgs49W5J1z