ChatGPT

Enhancing the Agentic Delivery Framework (ADF) for AI-Powered Development

Agentic Delivery Framework (ADF) – Overview and Goals

The **Agentic Delivery Framework (ADF)** is a vendor-neutral software development methodology designed for hybrid teams of humans and AI coding agents. It extends Agile/Scrum practices with additional guardrails so teams can integrate autonomous code generation safely. ADF preserves familiar Scrum roles – **Delivery Lead**, **Product Owner**, and **Developers** (which can include AI agents) – and mandates that all work goes through a governed planning and delivery flow [1]. In ADF, every code change is handled as a **Change Request (CR)** (e.g. a pull request) that must satisfy a set of quality gates (tests, security checks, etc.) before merge, ensuring each increment meets the Definition of Done and is fully auditable [1]. This approach tackles the unique risks of AI contributions by adding structure and oversight without abandoning Agile's iterative cadence.

Current Shortcomings in AI Development: Traditional "let the AI build my app" approaches often resemble a **waterfall** model – a single massive run of an agent trying to build an entire project – which is prone to multiple issues [2]. Common problems include **unsafe execution** (agents running code on unmanaged environments where changes are hard to track or revert), **opaque progress** (little linkage between the AI's actions and the original requirements), **environment drift** (agents working in sandboxes that don't match the team's actual tech stack), and **weak governance** (code being merged without rigorous reviews or policy checks) [2]. These issues make teams hesitant to trust autonomous coding. ADF directly addresses these pain points. It requires work to execute **inside a governed, auditable workspace** (never on a random developer's laptop), and it aligns agent activities with Agile artifacts like Product/Sprint Goals, user Stories, and Tasks [3]. Every Story in ADF produces a **Story Preview** (a pre-merge demonstration or spec of the feature) and daily **Pulse Increments** (daily integrated builds) so that progress is transparent and incremental [3]. Crucially, ADF enforces **automated checks and gates** on each CR – including tests, security scans, dependency audits, performance budget checks, and human approvals – to maintain high quality and compliance [3]. By speaking the language of enterprise process (Scrum roles, change controls, audit trails) and limiting Work In Progress (e.g. capping active Stories per agent), ADF ensures AI developers operate within safe bounds familiar to large organizations [3].

Key Features of ADF: At its core, ADF remains *Scrum-compatible* – retaining regular Sprint Planning, Reviews, Retrospectives, and Backlog Refinement – so teams can adopt it without disrupting their cadence [4]. On top of this, ADF layers novel practices tailored to AI: a **CR-first invariant** (all changes go through the Change Request process), a **Sequential Subtask Pipeline (SSP)** algorithm to break work into ordered subtasks with an exclusive "Story lease" (only one agent works on a Story at a time), daily **Delivery Pulse** meetings (a lightweight stand-up replacement focused on reviewing the latest AI-produced increment), and comprehensive **CR Gates** (the required checks before a PR merges) [5]. For example, ADF's default gates include `tests-ci` (all tests must pass), `security-static` (static analysis and secret scans), `deps-supply-chain` (dependency vulnerability checks), `perf-budget` (performance tests within limits), and even a `spec-verify` gate to ensure documentation and requirements are updated [5]. Only if all gates are green can the change merge; otherwise, the team must address the issues or apply a controlled "break-glass" override for emergencies [5]. This gate mechanism gives teams confidence that AI-generated changes won't slip through without proper

validation. The **Delivery Lead** (analogous to a Scrum Master) is charged with enforcing these processes – e.g. ensuring WIP limits, setting up the protected AI workspace, monitoring CI results – while the **Product Owner** ensures acceptance criteria are clearly defined and met (often via the Story Preview artifact) [6] [7]. Notably, ADF defines **multiple autonomy levels** to let organizations adopt AI gradually. At the basic level (L1 or "Supervised"), agents can only propose changes under close human oversight; at higher levels (L2 "Autonomy-Guarded" and L3 "Autonomy-First"), agents can work more independently (even 24/7) with stricter gating policies and limited human intervention, except for riskier changes [8]. This tiered model means even conservative enterprises can start with AI in a tightly controlled mode and increase autonomy as trust grows, while always maintaining required checks (e.g. L3 still demands SSP and all CR gates, using policy-as-code for enforcement) [8].

Enterprise-Ready Compliance: ADF was explicitly designed with enterprise compliance and auditability in mind. Every CR produces an **Evidence Bundle** – artifacts like requirements traceability, test results, security scan logs, and the Story Preview – that can serve as audit evidence [9]. An appendix in the ADF spec maps these outputs to common frameworks like SOC2 and ISO 27001, showing how, for example, a `requirements-trace.json` proving each requirement's implementation and approval ties to SOC2 CC7.2 (change management controls), or how attached preview/demo assets correspond to ISO 27001 A.8.34 for validating changes before release [10]. Similarly, ADF's built-in metrics align with **DORA** DevOps metrics (lead time, change failure rate, etc.), helping large organizations measure the impact of AI on their delivery performance [11]. By providing this mapping, ADF makes it easier for enterprises to adopt AI coding while still satisfying auditors and governance boards that proper controls (a "CAB-lite" change approval process) are in place [12]. In short, ADF's philosophy is **"Autonomy with Accountability"** – encouraging the productivity benefits of autonomous agents, but always with mechanisms to ensure accountability, traceability, and conformance to organizational standards.

SpecKit – Specification-Driven Development to Align AI Agents

SpecKit is a toolchain and methodology focused on *specification-driven development*, created to keep AI coding agents aligned with explicit requirements, architecture, and compliance rules. In practice, SpecKit encourages teams to write a single **structured requirements specification (SRS)** for their application or feature, and from that source it can generate various artifacts: documentation pages, test templates, an "agent brief" for the AI, and even an **RTM (Requirements Traceability Matrix)** to track coverage [13]. The idea is to prevent the AI (and the human team) from falling into what the author calls the "**vibe-coding**" loop – an undisciplined approach where one just prompts the AI ad-hoc without a clear plan or spec, leading to aimless iterations and mismatched outputs [14]. By contrast, SpecKit enforces a more rigorous approach: you define *what* needs to be built in a formal spec (potentially including security requirements, compliance checklists, etc.), and the tooling ensures the code and documentation stay in sync with that spec.

Core Features: SpecKit provides a CLI and TUI (terminal UI) that guide developers through creating and maintaining these specs. For example, you can initialize a new project from a template (including a "secure" preset template that comes with curated frameworks and compliance checks) [15] [16]. You then declare a *dialect* for your spec – for instance, SpecKit's own format (`speckit.v1`) or an industry standard like OWASP ASVS for security – which determines the structure of the spec and the validations applied [17]. Once the spec (often in YAML form) is written, a simple `speckit gen` command will generate human-readable docs (like a feature specification, an orchestration plan, and a "Coding Agent Brief" which distills requirements for the AI) [13]. These generated artifacts are stored in the repo (e.g. under `docs/specs/`) and serve as the single source of truth for requirements. SpecKit can also build an **RTM (Requirements Traceability Matrix)**, mapping each requirement (usually labeled `REQ-##`) to

evidence of implementation such as test cases or code references [18]. This ensures no requirement is forgotten and that every code change can be traced back to a documented need.

Importantly, SpecKit introduces automated **policy gates** to keep development on track. It integrates with version control and CI pipelines to continuously verify that reality matches the spec. For instance, running `speckit verify` in CI will *fail the build if any generated docs have "drifted" from the source spec*, prompting developers to regenerate docs or update the spec before merging [19]. This guarantees that documentation and code remain consistent. SpecKit also protects certain sensitive files: for example, it can require that changes to the spec catalog or compliance policies be label-approved and reviewed by code owners (using Open Policy Agent/Conftest rules) [19]. In practice, a CI setup might include: **(a)** a *SpecKit verify check* to ensure no spec drift, **(b)** a *catalog guard* that blocks unapproved edits to compliance templates, and **(c)** a *mode/policy guard* to prevent switching the project's mode (e.g. from secure to classic) or altering important security settings without oversight [19]. These controls mirror real-world governance (like preventing an AI agent from disabling its own safety features or bypassing a checklist). By enforcing such rules automatically, SpecKit builds confidence that an AI agent cannot silently ignore the spec or the organization's safety requirements.

Keeping AI on Track: One of the standout capabilities of SpecKit is its **Agent Run Coach and Forensics** features. As an AI coding agent works on tasks (especially if using an "autonomous" agent that plans and writes code iteratively), SpecKit can capture the *entire run log* – including prompts, the agent's thoughts or tool uses, errors, and outputs. It then analyzes these logs to produce structured insights. For example, the **SpecKit analyzer** will normalize raw agent logs into a `Run.json` and extract a list of all implicit or explicit requirements mentioned during the run (saving them to `.speckit/requirements.jsonl`) [20]. It computes various **metrics** about the run, such as *Requirement Coverage* (did the agent address the requirements it was given?), *Tool Precision* (how often the agent's tool/API calls succeeded), *Backtrack Ratio* (how frequently it had to retry or faced errors), *Edit Locality* (how concentrated were its code edits, indicating focus) and more [21]. The analyzer then produces a `summary.md` report and a `verification.yaml` – the latter containing a checklist of verifications (like tests or grep commands) that the agent should satisfy in the next iteration [20] [22]. Crucially, it also generates a `memo.json` of **guardrail instructions** learned from the run (for example, if the agent repeatedly made a certain mistake, the memo will note that) [23]. Developers can then run `speckit inject` to automatically merge those guardrails back into the **Coding Agent Brief** (the prompt context given to the agent), so that in the *next* run the agent is explicitly reminded of what to avoid or correct [24] [22]. This creates a **self-healing loop**: every agent attempt yields data that makes the next attempt smarter and safer. Over multiple iterations, the AI thus learns to better adhere to the spec and avoid known pitfalls, reducing the need for human re-prompting or intervention.

SpecKit's run forensics also feed into CI quality gates. For example, if the agent's run produced certain *forbidden failure modes*, SpecKit will attach labels to the run summary (like `process.read-before-write-fail` if the agent tried to edit code it hadn't loaded, or `env.git-state-drift` if the repository state changed unexpectedly) [22]. The CI pipeline can be configured to **fail the PR if these critical labels appear**, preventing merges when an AI run was fundamentally flawed [25]. Other metrics can be used to warn or set thresholds – e.g. requiring at least 75% requirement coverage and a tool success rate above some level for autonomous runs [26] [22]. All these artifacts (the run log, summary, RTM updates, etc.) can be stored as part of the Evidence Bundle for the change, further enhancing traceability. In summary, SpecKit acts as both **a compass and a guardrail** for AI developers: the upfront spec and generated brief guide the agent on *what to do*, and the analysis of each run provides feedback on *how well it was done*, enabling continuous improvement.

Security and Compliance Orientation: SpecKit also ships with **compliance "packs"** and secure presets to address industry regulations. For instance, the "secure" template mode can include mappings to standards like SOC2, GDPR, or HIPAA. SpecKit's design allows adapters for different dialects – one could use a security-centric spec dialect (like OWASP ASVS v4) so that the AI is explicitly instructed on security requirements from the start [27]. Additionally, SpecKit's internal governance (used in its own development) demonstrates best practices like preventing changes to security-critical configuration without special labels and tests [28] [29]. This philosophy aligns with ADF's emphasis on policy-as-code and gated changes. By using SpecKit, teams implicitly incorporate compliance requirements into their spec (rather than relying on after-the-fact code reviews), and they gain tooling that automatically checks those requirements at each step. The generated provenance metadata even tracks which "mode" (classic vs secure) and frameworks were used for generation, ensuring auditability of what standards were applied [30] [31].

In essence, SpecKit provides the **detailed blueprint and continuous inspection** needed to keep autonomous coding agents faithful to requirements and best practices. It complements a high-level methodology like ADF by focusing on the content of the work (requirements, specifications, compliance criteria) and by offering interactive tools to engage with the agent's process in real time. This makes SpecKit a powerful enabler for safe AI coding: it tackles the "prompt fatigue" problem by minimizing ad-hoc prompting (the agent always works off a structured brief and spec) and automates many tedious validation tasks that would otherwise fall to human developers. Indeed, **prompt fatigue** – the cognitive drain from having to iteratively craft and tweak prompts for AI [32] – is a recognized challenge in the industry, and SpecKit's answer is to front-load the effort into a spec (a one-time prompt design, so to speak) and then let automation enforce consistency thereafter. With the AI guided by a persistent spec and augmented by run-by-run coaching, teams can avoid constant micromanagement of the AI and instead focus on higher-level oversight.

Integrating SpecKit into the ADF Workflow

Although ADF and SpecKit were developed as separate initiatives, **they are highly complementary and can be integrated to create a more robust AI development lifecycle**. ADF provides the process framework (roles, when and how work moves, what gates to check) while SpecKit provides tooling and structure for the content (the requirements and code itself). Here's how they can work together and reinforce each other:

- **Spec-Driven Story Planning:** ADF already emphasizes having a clear Story definition and a Story Preview (demonstrable outcome) before calling a work item "Done." By using SpecKit during Backlog Refinement or Sprint Planning, teams can produce a detailed mini-spec for each Story *before* development starts. For example, the Product Owner and Delivery Lead might collaborate to write an SRS (Software Requirements Specification) for a new feature using SpecKit's YAML format, including functional requirements, non-functional constraints (security, performance budgets), and acceptance criteria. SpecKit can then generate a **Story Preview document** automatically from this spec, which ADF would treat as the required preview artifact for that story. This integration ensures that the "intent" (requirements) is crisply documented and fed directly into the development process. The **Coding Agent Brief** generated by SpecKit can serve as the starting prompt for the AI Developer working on the story, ensuring the agent is fully aware of the Story's requirements and context from the outset. This way, ADF's practice of Story Previews and SpecKit's practice of spec-first development become one and the same – the Story Preview *is* a spec document that the AI must satisfy. As a result, the linkage between intent and delivered artifacts becomes even more explicit (addressing the "opaque progress" issue): for every requirement in the Story spec, the eventual Evidence Bundle will contain proof (tests, code references via the RTM) that the AI's output meets it [10].

- **CR Gates Powered by SpecKit:** Many of the ADF's required Change Request gates can be directly implemented or enhanced using SpecKit's tooling. For instance, the `**spec-verify**` gate in ADF (which ensures method conformance and documentation updates) can leverage SpecKit's verify command to check that the documentation and spec have no drift [33]. If an AI agent made code changes that aren't reflected in the spec or if the spec was altered without regenerating docs, SpecKit will flag it, causing the `spec-verify` gate to fail until resolved [33]. Similarly, ADF's `**mode-policy**` gate (which enforces organizational policies like editing only allowed areas, or using approved frameworks) can hook into SpecKit's OPA-based checks. For example, SpecKit can label an agent's run with `edit-locality` metrics or flags if it strayed beyond the intended code scope [34] [35]. If the run shows the agent attempted to modify files outside the Story's domain (low *EditLocality* or a forbidden path edit), the `mode-policy` gate could fail, alerting the Delivery Lead to review that behavior. SpecKit's built-in *catalog protect* and *preset-change* checks are also directly applicable – they ensure that if the AI (or anyone) tries to alter the compliance templates or switch the project's safety mode, it triggers extra review [28]. These correspond to real-world governance: e.g., changing a security requirement would require security team approval. By integrating these into ADF's pipeline, we achieve **policy-as-code enforcement** of organizational rules, something ADF calls for at higher autonomy levels (L3 requires policy-as-code gates) [36].

- **Enhanced Evidence Bundles:** ADF mandates that each CR's Evidence Bundle include things like a requirements trace and test results [37]. SpecKit can automate much of this evidence collection. After an AI finishes a story and opens a Change Request, the team can run SpecKit's analyzer on the agent's run logs (which could be captured as part of the development workspace). This will produce a `requirements-trace.json` mapping each requirement (from the Story spec) to evidence in the code or test suite, which can be attached to the CR [38]. It also produces the `summary.md` of the agent's run and a metrics report; these can be attached or even posted as a comment on the PR for transparency [39]. In essence, SpecKit becomes the engine that assembles the Evidence Bundle. The advantage is consistency and thoroughness: instead of relying on a human developer to manually write a summary or show proof that all acceptance criteria are met, the system generates a neutral, factual record. This is invaluable for enterprise audit trails – for example, the mapping of requirements to code satisfies auditors that no requirement was implemented without approval and testing [10]. It also streamlines the Product Owner's validation: they can read the Story's spec and the attached Story Preview/summary to quickly verify that what the AI built matches what was intended.

- **Continuous Improvement via Delivery Pulse:** ADF's Delivery Pulse is a daily sync where the team inspects the latest integrated work (Pulse Increment) and evaluates any risks or issues [40] [41]. By incorporating SpecKit's metrics into this Pulse, the team can quantitatively track how the AI agents are performing. For example, each day the Delivery Lead could report the AI's **ReqCoverage** (requirements coverage) and **BacktrackRatio** from the previous day's runs, which are indicators of how smoothly the AI is working [21]. If coverage is low or a lot of backtracking occurred, that might prompt adjusting the approach (perhaps the spec needs to be clarified, or the agent is struggling with a particular task and needs a new strategy). Over time, these metrics can serve as *leading indicators* of problems even before gates fail. Additionally, SpecKit's **failure labels** (like the read-before-write or state drift) could be aggregated in the Pulse to highlight pattern failures. Delivery Pulse meetings might include reviewing the SpecKit-generated PR summary for any AI-run anomalies. Essentially, ADF provides the meeting and cadence to do this, and SpecKit provides the data to make those meetings rich and focused on improvement rather than guesswork. This tight feedback loop can greatly increase confidence in AI contributions: the team isn't waiting for a big surprise at merge time; they are continuously watching the agent's learning curve and intervening early if needed.

- **Autonomy Level Transitions:** When combining ADF and SpecKit, teams have a clearer path to safely increase AI autonomy. At **Autonomy Level A1** (assisted execution with human in loop) or A2 (supervised autonomy with human approval on merges) [42], SpecKit can be used to its full extent to catch issues, but a human is still approving merges. The data gathered by SpecKit in these stages (like how often the AI needed corrections, how often gates passed on first try, etc.) can inform when a team is ready to try **A3 (budget-constrained autonomy)** [43]. By the time they consider letting the agent merge code on its own (no human approver for each PR), they will have a backlog of evidence showing the agent's reliability metrics. SpecKit's gating can then act as the safety net in lieu of a human: if an agent tries to merge code that violates a policy or drags down quality, the CI `speckit-pr-gate` will stop it [25]. In A3/A4 modes, ADF requires additional gates like `cost-budget` (limiting AI spending) and `lease-broker` (ensuring exclusive control of a story) [44] [45]. SpecKit can assist here as well – for example, the spec for a Story could include a cost budget which the agent is aware of, and logs from the run (or the cloud API usage data) could be checked against this. For the lease-broker, an **Agent Run Ledger** (a tamper-evident log of all actions the agent took) is needed [46]; SpecKit's detailed logging and even the hash-chaining of its outputs could contribute to building that ledger. By integrating at this level, **even at high autonomy, the AI is never "unchecked"** – it's constrained by automated guardrails at every step, which is exactly what conservative enterprise change management demands.

In summary, integrating SpecKit into ADF can make the methodology **even more rigorous yet efficient**. The spec-driven approach ensures the AI has a clear contract to fulfill, aligning with ADF's emphasis on Definition of Done and acceptance criteria. The continuous verification and analysis that SpecKit provides fill in the operational gaps – giving real-time insight into the agent's behavior and ensuring that all ADF gates (and more) are enforced with minimal human toil. This synergy directly addresses the concern of **"prompt fatigue" and oversight burden**: instead of a human having to babysit an AI with constant prompts and code reviews, the combination of ADF process + SpecKit automation lets the AI run within a controlled sandbox, with the methodology catching any deviance. Developers can trust the process because any misalignment will be caught either by a failing test, a failing SpecKit check, or a metric trigger, much like a failing unit test would catch a regression. By using SpecKit's outputs as part of ADF's required artifacts, we essentially teach the autonomous agent to become a disciplined Scrum team member – one that reads the spec, follows the checklist, and documents evidence of its work just as a human professional would.

Recommendations to Evolve ADF for Broader AI Adoption

While ADF (augmented by SpecKit) is already a cutting-edge framework, there are further improvements and research insights that can make it even more beneficial for teams – especially large enterprises – looking to safely embrace AI in their development cycle. Below are key recommendations and enhancements for the Agentic Delivery Framework:

- **Promote Specification-Driven Development as a First-Class Practice:** To encourage consistency and reduce ambiguity, ADF should explicitly incorporate spec-driven workflows in its guidance. This means advising teams to begin each Epic or Story with a clear, formal specification (possibly using tools like SpecKit or similar). By making "Write the Story Spec" a standard step before coding, we ensure that AI agents are not operating on fuzzy requirements. This will prevent the drift of scope and "vibe coding" where developers or agents improvise features without firm guidelines [14]. In practice, ADF can include templates for Story specifications and acceptance criteria (extending the current Story Preview template) that map to compliance requirements as needed (e.g. fields for security considerations, performance

budgets, etc.). This would amplify ADF's existing rule that there must be a Story Preview for every Story – by turning that preview into a comprehensive spec/plan, the agent's work remains *faithful to requirements and existing design constraints*. It also reduces prompt fatigue by front-loading context: once the spec/preview is written, the AI can be given this document and doesn't need repeated prompting for details. Encouraging the use of spec generation tools in ADF's adoption guide will help teams new to AI get started on the right foot, avoiding trial-and-error prompting and instead focusing on writing good requirements (which is a familiar skill in traditional development).

- **Leverage Multi-Agent Role Specialization:** Research and experiments are increasingly showing that a **multi-agent approach** – where different AI agents assume roles like Developer, Tester, Reviewer, etc. – can significantly improve outcome quality [47]. ADF's roadmap already hints at this (e.g. a "test agent, docs agent, fixit agent" in future plans) [48]. Formalizing this, ADF could define patterns for multi-agent collaboration within a Sprint. For example, once a Coding Agent (Developer role) finishes a change, a Testing Agent could automatically generate additional test cases or run exploratory tests on the new code, and a Reviewing Agent could perform an initial code review against the spec and coding standards. These agent roles would operate under the same gated process (e.g., the Test Agent's output must pass in CI, the Reviewer Agent's feedback might be attached to the PR). By splitting responsibilities, each agent can focus on a narrower task with clear success criteria, which aligns well with how Agile teams already separate concerns (development vs QA vs code review). This specialization is supported by recent frameworks like *AgileCoder*, which successfully mimic an Agile team with multiple LLM-based agents handing off tasks iteratively [49] [50]. ADF could incorporate guidelines on orchestrating such agents, perhaps via the Delivery Lead's orchestration script (P1 in the roadmap) – e.g., after a story's code is drafted by the Dev Agent, automatically trigger a Test Agent to validate it and a Doc Agent to update documentation. Human team members would then supervise the interplay and only intervene on exceptions. This can dramatically increase productivity (parallelizing work) while keeping quality high, since the agents check each other's work. It also has enterprise appeal: it mimics the segregation of duties common in large organizations (developers vs testers vs auditors), but at machine speed.

- **Strengthen AI Safety Rails with Static Analysis and Knowledge Graphs:** To handle the complexity of large codebases, ADF could integrate more advanced static analysis and context-provision techniques for AI agents. One idea is to use a **code dependency graph or knowledge base** so that the agent always has relevant context for the part of the system it's working on. For example, *AgileCoder* introduced a dynamic Code Graph Generator to help agents retrieve pertinent code context in each sprint, rather than relying purely on the LLM's memory [51] [52]. ADF could recommend establishing an internal "code wiki" or vector database of the codebase that agents can query (with embeddings for code semantics), ensuring they don't hallucinate functions that exist or overlook important modules. This directly addresses issues of an AI not understanding the entire system (a common shortcoming noted in AI coding research) [53]. By equipping the agent with query tools to explore the codebase (and by extension, architecture docs or past decisions), we reduce the chance it goes off track or introduces inconsistencies with the existing implementation. Concretely, a **safety rail** could be added that whenever an AI agent is about to implement a function, it must first search for related code or existing utilities (something SpecKit's run hints already encourage, like suggesting the agent to grep the repo before editing unfamiliar files). This could be enforced by a policy that flags if an agent wrote code that duplicates existing functionality. Including such static analysis and context retrieval steps in the SSP (Sequential Subtask Pipeline) guide will make AI contributions more **faithful to the existing implementations** and architecture, a concern the user highlighted.

- **Introduce "Prompt Hygiene" and Verification Checklists:** Prompt fatigue can also be mitigated by improving prompt quality and reusability. ADF could incorporate a concept of **prompt hygiene** – essentially, maintaining high-quality, version-controlled prompts or agent instructions as part of the project artifacts. For example, the "Coding Agent Brief" generated via SpecKit (which contains the distilled spec and guardrails) should be stored and treated as code. ADF might prescribe that this brief be updated whenever requirements change, and peer-reviewed just like code or design docs. Additionally, building on SpecKit's verification checklist, ADF could mandate that every Story has an accompanying *verification plan* (a list of things to verify or tests to run, possibly auto-generated) that the agent must satisfy before calling the task done. The agent then uses this as a to-do list internally. This idea pushes more responsibility onto the AI to self-check its work, reducing human verification effort. It resonates with the concept of agents reflecting on their outputs. Recent trends (like the "Reflexion" approach for agents) have agents generate checks for themselves and iterate until those checks pass. Making this an official part of the methodology (e.g., "**Definition of Done for AI**: the agent must confirm all items in the verification checklist are resolved, and include evidence for each in the CR description or Story Preview") will institutionalize a self-QA behavior in AI agents.

- **Fine-Tune Autonomy with Granular Trust Metrics:** As organizations get comfortable with AI, they will want to quantify when an agent is ready for more autonomy. Beyond the binary gate pass/fail, ADF can incorporate **metrics-driven trust thresholds**. For instance, require that for an agent to operate in Autonomy Level A3 (merge on its own at night), it has demonstrated over the last N sprints a >90% test pass rate without human fixes, <5% reopens of its PRs due to defects, and no security incidents. These kinds of stats can be collected via SpecKit and CI (some are analogous to DORA metrics adapted for AI). ADF could define a "readiness checklist" for moving to each higher autonomy level, including both process compliance and performance metrics. This provides a *safety gradient*: enterprises can adopt AI coding in a staged way, with data to back each promotion in autonomy. Additionally, ADF might integrate a **kill-switch or rollback plan** in the methodology for high-autonomy agents: e.g., if certain error rates or metrics regress, the Delivery Lead must temporarily revert to a lower autonomy mode (A2 or A1) until issues are resolved (kind of like automatically tightening guardrails when anomaly detected). This dynamic adjustment keeps risk in check.

- **Enterprise Integration and Culture Change:** Finally, to drive adoption in large organizations, ADF should include guidance on the organizational side: training, culture, and roles. For example, the **Delivery Lead** role could be expanded in description to highlight new responsibilities like monitoring AI ethics (ensuring the AI doesn't introduce biased or licensed code) and serving as an **AI operations coach**. We should also consider establishing an **ADF Alliance (Community)** – akin to the Agile Alliance – which the user mentioned as an aim. This could steward the framework's evolution, certify practitioners, and share case studies from industry. To this end, improving the methodology might involve creating an official **ADF certification program** or **knowledge base** that enterprises can tap into, giving them confidence that adopting ADF isn't a leap into the unknown but a well-supported transition. Including references to how ADF maps onto existing enterprise processes (e.g., how it fits into ITIL change management, or how it can work with common tools like JIRA, Azure DevOps, etc.) will also smooth adoption. Large companies will appreciate concrete mappings like "a Story in JIRA corresponds to a Story + Story Preview in ADF; here's how to implement that in your ticketing system," or providing sample configurations for GitHub/CI (many of which already exist in the ADF examples). Essentially, **meet enterprises where they are**: show that ADF doesn't upend their compliance or project tracking, it augments it. Emphasizing the success metrics (e.g., 0 direct pushes, <24h lead time for stories, daily demos) [54] as achievable outcomes will help

make the case that adopting ADF is not just about AI safety, but also about tangible improvements in speed and quality.

In conclusion, by **integrating SpecKit's spec-driven rigor, adopting multi-agent strategies, and incorporating the latest research and metrics**, the Agentic Delivery Framework can be continuously improved into a comprehensive methodology that large enterprises can trust. These enhancements aim to eliminate the remaining pain points (like prompt fatigue and uncertainty about AI outputs) and replace them with structured, reliable processes. The end vision is very much aligned with the user's goal: a future **"Agile Alliance" for AI development**, where companies of all sizes have a clear blueprint for incorporating autonomous coding agents into their teams **safely, securely, and productively** – not as a novelty, but as a standard practice. By addressing both the technical guardrails and the human processes (roles, training, audit), ADF can help organizations unlock AI-driven productivity while **maintaining confidence that every change is on track and compliant with requirements**. This melding of Agile principles with AI-specific practices could become the industry's de facto playbook for AI-augmented software teams, much like Scrum did for traditional teams. With continuous research, open collaboration, and the improvements outlined above, ADF can evolve into that robust, enterprise-ready framework that ensures autonomous coding agents truly become a safe asset rather than a liability in modern development workflows.

Sources:

1. Gunning, A. *Agentic Delivery Framework (ADF) — Methodology* (v0.5.0), *airnub/agentic-delivery-framework-internal* [1] [2]. Describes the ADF approach, roles (Delivery Lead, Product Owner, AI/Hybrid Developers), and the CR-first, gated workflow for safe AI-human team software delivery.
2. Gunning, A. *ADF Overview and Problem Statement*, *airnub/agentic-delivery-framework-internal* [2] [3]. Highlights shortcomings of naive "AI builds app" attempts (waterfall-like, unsafe, ungoverned) and outlines ADF's vision of governed workspace, observable Scrum workflow, and guardrails (Story Previews, Pulse, automated reviews).
3. Gunning, A. *ADF CR Gates and Evidence*, *airnub/agentic-delivery-framework-internal* [38] [10]. Defines the required quality gates (tests, security, spec verification, etc.) on every Change Request and maps ADF's evidence outputs to compliance frameworks (SOC2, ISO27001) and DevOps metrics (DORA), ensuring enterprise auditability.
4. Gunning, A. **SpecKit – Getting Started**, *airnub/speckit-internal* [13] [19]. Explains SpecKit's spec-driven development flow: generating documentation and an RTM from a single spec, and using CI gates (no drift, protected catalogs, policy guardrails) to keep AI coding on spec and out of the "vibe-coding" trap.
5. Gunning, A. *SpecKit Agent Run Forensics*, *airnub/speckit-internal* [20] [22]. Describes how SpecKit collects AI agent execution logs and analyzes them to produce run artifacts (requirements coverage, errors, metrics) and inject learned guardrails back into the agent's prompt. This ensures each autonomous run improves on the last and critical failures (e.g. environment drift) are caught in CI.
6. Xu et al. **"AgileCoder: Dynamic Multi-Agent Framework for Agile Software Development"**, *arXiv preprint 2024* [50] [47]. Academic work contrasting waterfall-style AI coding (ChatDev, MetaGPT) with an Agile, multi-agent approach. Demonstrates that assigning LLM-based agents distinct Scrum roles (programmer, tester, reviewer) and iterating in sprints yields better code quality and efficiency, validating ADF's multi-agent vision.
7. Frank, E. *"Burned out by bots: The rise of prompt fatigue in the workplace"*, *Computerworld* (Sep 2025) [32]. Industry article documenting *prompt fatigue* as a real phenomenon – the cognitive exhaustion from constantly refining AI prompts. Underscores the need for frameworks like ADF+SpecKit that minimize continuous prompt tinkering by establishing upfront specifications and automated feedback loops.

1 4 5 6 8 36 40 41 README.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/README.md

2 3 7 48 54 overview.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/docs/overview.md

9 37 glossary.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/docs/glossary.md

10 11 12 appendix-enterprise-mapping.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/docs/specs/appendix-enterprise-mapping.md

13 14 15 16 17 18 19 getting_started.md
https://github.com/airnub/speckit-internal/blob/e1d025e1168327e3ca886134edfd3ddd2fd77fb0/docs/getting_started.md

20 22 23 24 26 speckit-run-forensics.md
https://github.com/airnub/speckit-internal/blob/e1d025e1168327e3ca886134edfd3ddd2fd77fb0/docs/speckit-run-forensics.md

21 25 39 speckit-run-coach.md
https://github.com/airnub/speckit-internal/blob/e1d025e1168327e3ca886134edfd3ddd2fd77fb0/docs/speckit-run-coach.md

27 28 29 30 31 mode-assurance.md
https://github.com/airnub/speckit-internal/blob/e1d025e1168327e3ca886134edfd3ddd2fd77fb0/docs/internal/charters/mode-assurance.md

32 Burned out by bots: The rise of prompt fatigue in the workplace – Computerworld
https://www.computerworld.com/article/4047909/burned-out-by-bots-prompt-fatigue-in-workplace.html

33 34 35 38 44 45 cr-gates.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/docs/handbook/cr-gates.md

42 43 46 appendix-autonomy-levels.md
https://github.com/airnub/agentic-delivery-framework-internal/blob/fb1b3ab19903b5d61c98af733cc56fecc17265b2/docs/specs/appendix-autonomy-levels.md

47 49 50 51 52 53 : Dynamic Collaborative Agents for Software Development based on Agile Methodology
https://arxiv.org/html/2406.11912v1