NVISIA SOLUTIONS

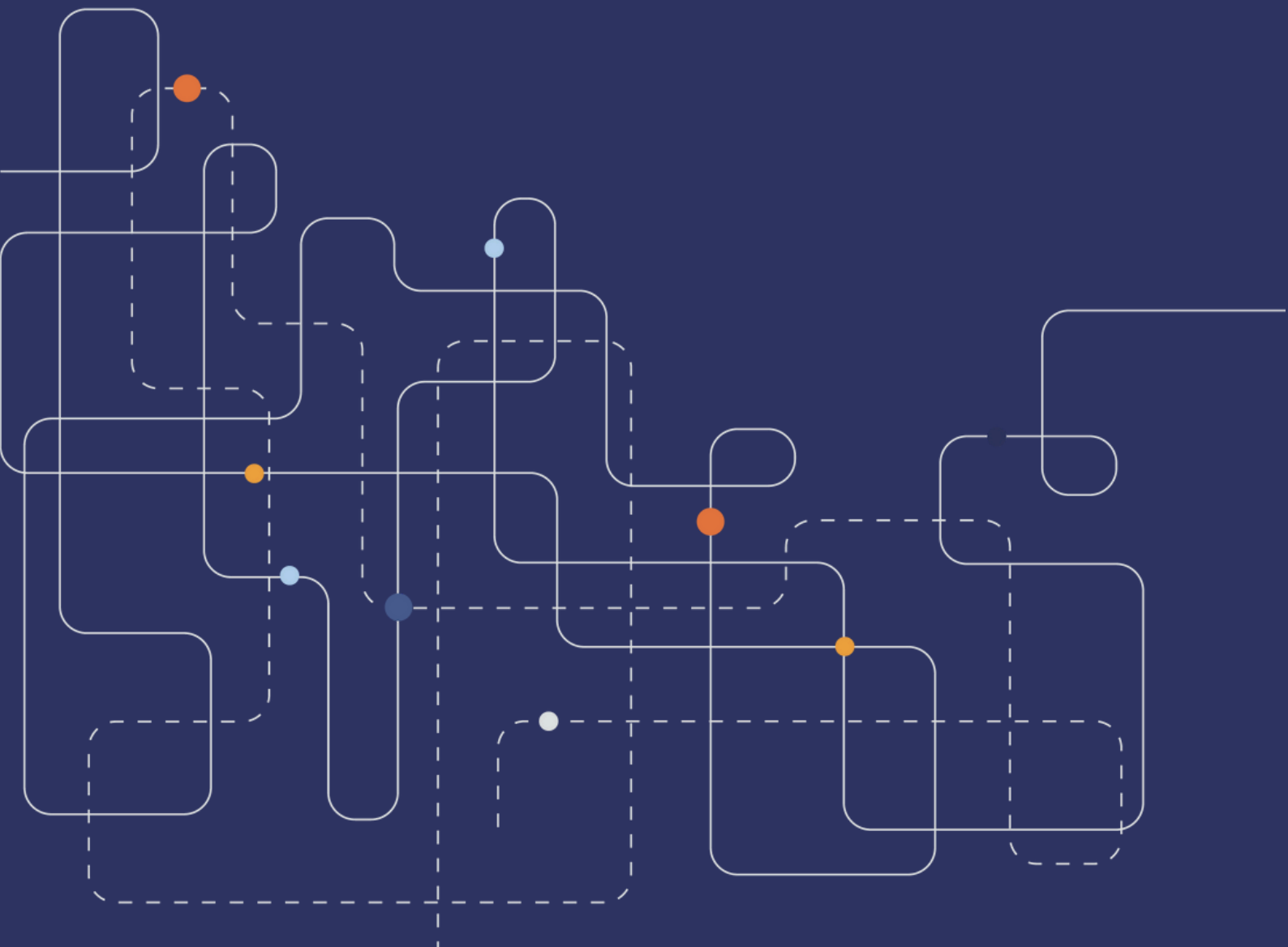
nvisia
connect. build. enable.

Spec-Driven Development

My Journey Using the GitHub Spec Kit

MICHAEL HOFFMAN
nvisia Technical Director

NOVEMBER 2025



Introduction

Additional Report Context and Insights

For the past few months, I've been assisting engineering teams with the adoption of AI Coding Agents. In my previous article, "Prompt Engineering Management: Scaling AI Guardrails and Collaboration via Shared Prompt Libraries", I shared my initial experiences with turning individual prompt engineering expertise into a shared, governed asset. This included using GitHub Copilot Instructions to define non-negotiable guardrails and standards as part of context. It also covered templates as a scaffold for ensuring more consistent, high-quality output from the AI Coding Agent.

Recently, I've been exploring Spec-Driven Development (SDD) and the GitHub Spec Kit as a potential answer to the governance gaps left by ad-hoc prompt engineering. SDD uses an executable specification as the source of truth for feature development, rather than relying solely on application code or WIKI content. This approach reflects an industry shift towards Context Engineering and Agentic AI, with proponents envisioning SDD as the enterprise-grade approach for AI-first development.

I'll offer an honest assessment of this emerging approach. While the promise of SDD is compelling, I believe it's currently too early for enterprise-grade production adoption. Challenges like over-specification bottlenecks, usability friction, and spec drift can severely impact delivery and reliability. I encountered these issues first-hand while using the GitHub Spec Kit to build a proof-of-concept transaction fraud detection platform. However, these challenges do not render the approach useless; I will share my insights into where I believe SDD could be useful, and the critical lessons learned that can guide your exploration toward success.

Why Spec-Driven Development?

In his article, "Spec-driven development with AI", Den Delimarsky shares his definition of SDD:

> **Instead of coding first and writing docs later, in spec-driven development, you start with a (you guessed it) spec. This is a contract for how your code should behave and become the source of truth your tools and AI agents use to generate, test, and validate code. The result is less guesswork, fewer surprises, and higher-quality code.**

The core idea is that the specification becomes a living, executable artifact that drives the entire development lifecycle. So, why does this matter? SDD proponents claim this approach delivers three significant benefits by changing how we develop:

- **Reliability and Quality:** SDD provides enforceable intent and governance upfront in the context of prompts, leading to more reliable and higher-quality task execution by AI Coding Agents.
- **Simplified Iteration and Refactoring:** Separating the "what" (the spec) from the "how" (the code) allows humans to update intent, which AI Agents then use to more efficiently implement complex code changes and iterative development.
- **Focused, Smaller Development Teams:** Shifting responsibilities creates focused teams who become "steerers and verifiers," delegating code writing to the AI Agent and focusing instead on defining intent and validating the output.

Addressing Knowledge Fragmentation

Beyond direct development benefits, SDD addresses a foundational organizational problem: knowledge fragmentation.

Crucial context and architectural decisions for projects are often scattered across multiple platforms, including Slack, Teams, email, user stories, and internal WIKIs. Due to tight deadlines, this vital information is inevitably lost or siloed, resulting in a direct impact on business operations:

- **Expensive Onboarding:** New employees spend weeks or even months searching for system

context, leading to lost productivity.

- Increased Project Time and Cost: Projects necessitate expensive discovery phases, bringing in subject matter experts to manually reconstruct how systems work before any coding can begin.

- Increased Maintenance Cost: The time required for support staff to investigate root causes and address issues spikes, impacting business continuity and customer satisfaction.

Treating the specification as a long-lived, centralized source of truth could drastically reduce the time developers spend searching for information and context. This leads to faster development cycles, more consistent system behavior, and significantly less friction when bringing new team members up to speed.

Ultimately, I see SDD as well-aligned for emerging practices in Context Engineering. As AI Coding Agents perform tasks, having structured specifications and governance as part of the context should radically improve the chances of an LLM returning the desired, compliant results.

The Core Problem SSD Addresses

My experience with Prompt Libraries confirmed the need for shared context but exposed a fundamental limit: a prompt is just an instruction; it lacks the formal, machine-readable structure required to strictly enforce organizational standards. Crucial context (architectural decisions, security policies, etc.) remained scattered across corporate platforms.

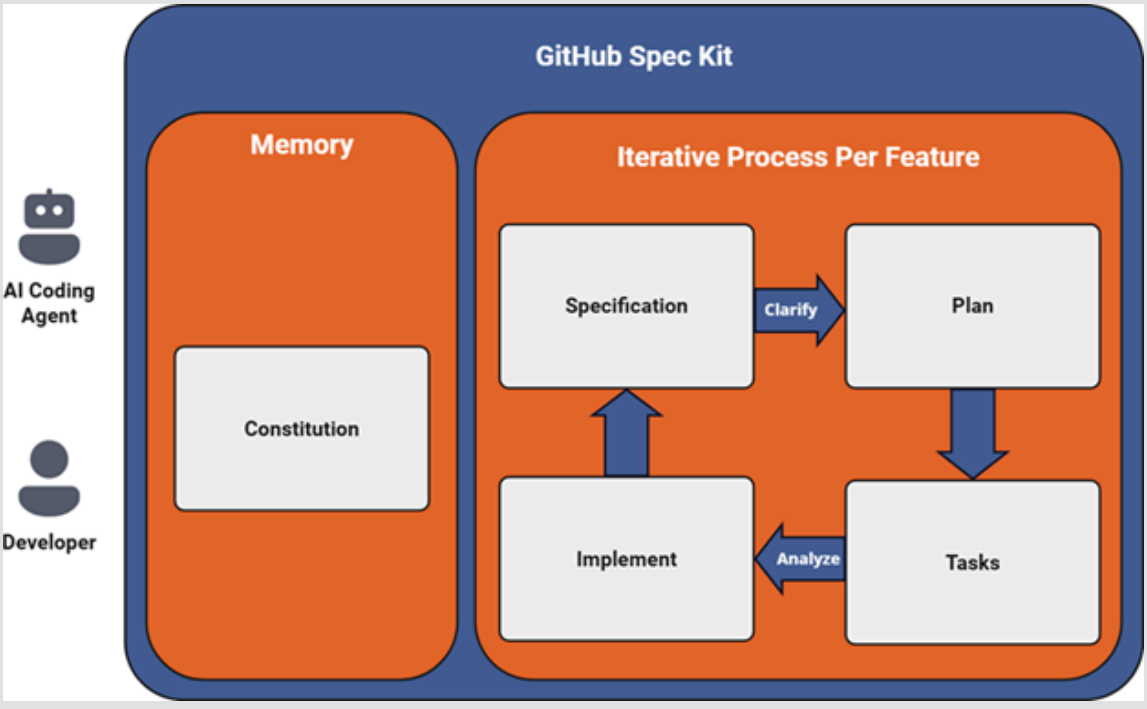
SDD attempts to solve this by providing a framework where the Specification becomes the ultimate, version-controlled shared context, enforceable by the AI Agent.

Introducing the GitHub Spec Kit

To date, several tools have been released to support Spec-Driven Development. In this article, I'll be focusing solely on the GitHub Spec Kit. If you are interested in other options, check out either Kiro or Tessl, both of which provide a full agentic development environment as well as support for SDD. The GitHub Spec Kit was released in August 2025. Currently, it provides you with several groups of artifacts:

- **The Specify CLI:** Gets you started with a new project leveraging the spec kit. The command will generate the initial repository, creating templates and prompts that are tailored to the AI Coding Agent and operating system of your choice.
- **Prompt Files:** Supports executing a prompt with a given AI Coding Agent for each step of the spec kit's workflow.
- **Templates:** Defines the structure of key artifacts, including the Specification and Constitution.
- **Scripts:** Execute commands from a command line, such as for interacting with Git.

The GitHub Spec Kit operationalizes SDD by enforcing a rigid, four-phase, gated process flow, as shown in the diagram below:



The key to governance lies in the **Constitution** artifact, a non-negotiable set of rules (Development Standards, Technical Constraints, Governance Rules) stored in the project's **Memory** folder. Once the Constitution is defined, the developer follows an iterative process for each feature:

1. **Specification (The What):** Define business requirements, user journeys, why the feature is being built, and what are the expected outcomes.
2. **Plan (The How):** Convert the specification into a technical architecture, detailed deisign, and implementation strategy.
3. **Tasks (The Steps):** Break the plan into sequential, atomic tasks for the AI agent to execute.
4. **Implement (The Code):** The AI Agent executes the tasks in order while the developer verifies the output from the Agent.

The developer's role fundamentally shifts from writer to **steerer and verifier**, ensuring the artifacts meet quality standards at each step as the **human-in-the-loop**. During the process, the developer also can use prompts to **clarify** the specification in case there are ambiguities and **analyze** the specification, plan, and tasks for clarity prior to implementation.

Evaluating the GitHub Spec Kit

For evaluation, I chose to build a Fraud Detection Platform using the GitHub Spec Kit. The Fraud Detection Platform was for an imaginary organization, GloboBank. This type of platform seemed interesting as a use case with both front-end and back-end concerns to address.

Creating the Project: One Project or Multiple?

While the first step should seem obvious, SDD brings in some interesting questions as you get started with your first project. If I have an enterprise project, it's likely spread across multiple repositories. For example, the Fraud Detection Platform I plan to build would have separate repositories for the user interface and backend services. I began to wonder,

- Should **each repository track their own specification**?
- Should there be a **single, shared specification**?
- If it's shared, how do the different repositories **include it in context**?

While there wasn't much guidance in the Spec Kit documentation, there is an active discussion about this topic on the project's discussion board.

In my opinion, it seemed like a better approach to keep a centralized repository specification containing the high-level business requirements across components.

This centralized specification is still managed by GitHub Spec Kit but never moves past the specification and clarification phase. Then, each component has a **technical specification** managed by GitHub Spec Kit rather than a functional or business specification.

There's a consequence to this approach as it requires more work keeping the centralized specification in sync, leading to potential **Spec Drift**. Additionally, the AI Coding Agent needs access to this centralized specification to use it in the context of prompts. I'll be interested to see how others approach this problem.

If you'd like to see the result of this structure, you can view the three repositories I created on GitHub:

- **GloboBank Fraud Detection Platform:** Repository for the master specification for the platform

- **GloboBank Fraud Service:** Repository for the back-end fraud service.
- **GloboBank Fraud Management UI:** Repository for the User Interface.

Creating the Constitution: How Do You Right-Size Governance?

Once I knew my project structure, the next step was to create the Constitution file. The constitution defines the non-negotiable governing principles and development guidelines for the project.

What should the content of this file include? Here are some examples:

- **Development Standards:** What are the requirements for code quality metrics? What are the minimum technical requirements for response time of web pages?
- **Technical Constraints:** What is the standard cloud provider? Which frameworks and versions are used for services? What are the security compliance requirements for data?
- **Governance Rules:** How often should the constitution be reviewed? What is the process for amendments to the constitution? Is there approval required for amendments?

Ideally, this content is centralized and standardized to be pulled into the Constitution automatically by the AI Coding Agent; however, in most cases, the Constitution will be derived from several sources, such as Centers of Excellence, product teams, and operations. It's expected that the developer will own the generation and update of the constitution; however, the AI Coding Agent can be leveraged for acceleration and verification as needed.

As I started to create the Constitution file, I saw potential challenges. Each repository could have different principles. For example, a back-end service probably won't care about UI standards, and vice versa. There's also a question of scalability as multiple having hundreds, or even thousands, of Constitution files strewn across repositories will quickly become unmanageable.

I've been exploring the concept of Knowledge Bases to use in Context Engineering, and I think it could be useful for building Constitutions.

Knowledge Bases define rules, best practices, etc. in Markdown files that can be added to the context of a prompt. These Knowledge Bases can be at different hierarchical levels and for different disciplines. Let's look at an example in the GloboBank Knowledge Base I created.

- At the top level is the **GloboBank Knowledge Base**. This defines the organization's core values, compliance, financial standards, etc. An example of Compliance and Regulatory Adherence is: "Compliance with GDPR, PCI-DSS, and other data protection regulation".

- At the next level are **Disciplines / Communities of Practice Knowledge Bases**. This includes governance from architecture, product, security, cloud, DevOps, etc. Each area would own their updates to their specific Knowledge Base.

- The third level is the **Domain Knowledge Base**. An example is the Fraud Domain, which owns standards and practices specific to their domain. These could extend, refer to, or even override higher-level Knowledge Bases.

The approach I took was to use this knowledge base for generating a project-specific Constitution file. In my IDE, I included the knowledge bases folder; though, you could also use other approaches, such as a GitHub MCP server for the AI Coding Agent to read your knowledge base repository. At regular intervals, the team can refresh their Constitution and amend it through an analysis against the centralized knowledge base.

If you'd like to see the result of generating these constitutions through GitHub Spec Kit's prompt, just click the links below for examples:

- The GloboBank Fraud Detection Platform's Constitution

- The Fraud Management User Interface Constitution

- The Fraud Service Constitution

Creating the Specification

With the Constitution in place, it was time to create my first feature specification. The first specification I tried was creating the transaction scoring rules engine. I quickly disc-

-overed that the AI Coding Agent will generate significant amounts of specification text if the feature spec is too broad. This wall-of-text was overwhelming, leading me to greatly reduce the scope of my first spec. I prompted for implementing a single, simple rule to score fraudulent transactions.

Here's where I see both potential value and challenges with the step:

- **Constitution Considerations:** Because the constitution check is in place, the specification included related details, such as regulatory requirements. As a business analyst, this greatly reduces cognitive load with the initial creation of requirements.

- **Specification in Business Language:** The specification language keeps communication at the business level without diving deep into technical implementation. This can help steer business and product away from technical discussions and maintain focus on the "what" and not the "how".

- **Potentially Easier Transition to Agile Tools:** While I haven't evaluated this yet, the centralized specifications could be turned into an epic or feature for tracking in a tool like Jira or Azure DevOps. This could even be automated with MCP.

- **Non-Standard Template:** There could be challenges, either in an organization or even in a team, where specifications have different structure. I could envision templates becoming part of governance and the Constitution, defining key sections, like the business context and acceptance criteria.

- **Ownership:** A key question I began asking is: who owns this specification? Is it the product owner? Is it an engineer? If it's intended to be the source of truth, then a separate copy shouldn't be managed in outside documentation. Managing a specification in a GitHub repository may make sense for engineers but might present challenges for non-technical folks.

You can find the feature and technical specifications I created at the following links:

- GloboBank Fraud Detection Platform Spec

- GloboBank Fraud Management User Interface Spec
- GloboBank Fraud Service Spec

Planning, Tasking, and Implementing a Feature

The GitHub Spec Kit defines a rigid workflow for development. Once the specification is considered "done", the engineer(s) can then follow the steps of planning, creating tasks for the AI Coding Agent, and then directing the AI Coding Agent through implementation. This is where I began to discover usability friction.

For example, the Constitution has a guardrail for using Kafka. As a result, the technical specification for my simple rules engine was quickly expanded by the AI Coding Agent. The challenge here is that if I find architectural flaws late in the implementation phase, I may have to roll back up to the technical specification, or even the feature specification.

Another challenge I encountered was an over-specification bottleneck. This surfaced in several ways. Going back to the example of Kafka, I had to clarify in the specification that I would address streaming in a future specification.

This wasn't the only instance though. I needed to punt on security, logging, deployment to AWS, etc., all because I needed to circumvent governance for this simple feature.

Another way this challenge surfaced was in the frequent conversational prompts required to "get it right". This is where being the human-in-the-loop can become frustrating. Do you detail every last aspect of the project in the specification?

At that point, is it easier to just create the code manually? For larger feature sets, this conversational approach could be beneficial, but for smaller, quicker fixes, it may not be.

Lessons Learned and Closing Thoughts

Coming out of this experience, I can share a few additional lessons learned and thoughts with you:

Use Conversational Prompts

I found that requesting the AI Coding Agent to be in a conversational mode was much easier as the human-in-the-loop analyzer. I could understand and ask questions in each step of the workflow. This is an improvement over reviewing the wall-of-text from an AI Coding Agent response or reviewing the update to several artifacts all at one time. You could even add this to the Constitution so it doesn't have to be defined per prompt.

Challenge of Initial Enablement

A potential challenge with SDD and the GitHub Spec Kit is the higher amount of time required for enablement in the initial specs. This issue would exist even in traditional development, but I could envision engineers feeling that the Agent is taking more time because of the challenges faced in initialization and architecture enablement. This includes database configuration, security configuration, etc.

Commit Changes to Version Control Frequently

Another good practice is to commit your changes frequently to Git. Any time a file is created or updated, the spec kit uses versioning inside the file contents. If you don't commit often enough, you risk losing an iteration / version of the document. This can also help if you want to try the AI Coding Agent prompts against different models.

Potential Use Cases

It feels like SDD and the GitHub Spec Kit provide a significant improvement over Vibe Coding, making it viable for usage in rapid prototyping. While I've yet to try the approach for an existing application, I could immediately see several challenges when compared to greenfield development. The most significant challenge is the existing knowledge segmentation and information silos. By reverse engineering the code into specs, it's unlikely that the context will be correct in many situations. It could be possible to use MCP and access Jira for reading epics, features, and stories, but that assumes the information in

these artifacts are up to date and accurate. Over time, these stories become stale and out-of-date, potentially resulting in inaccurate information within the specs.

Overall, my journey with SDD and the GitHub Spec Kit was a valuable, though sometimes frustrating, glimpse into the future of development with AI Coding Agents.

While SDD seems promising, the current tooling is not yet ready for enterprise-grade production. The best way to understand this new workflow is to see it for yourself. I encourage you to explore the three GloboBank repositories I linked in the article to see the full structure, constitutions, and specifications from my experiment.

My work is focused on helping engineering teams navigate this exact transition. If your organization is wrestling with AI governance, scaling prompt engineering, or developing a strategy for AI-first development, I'd love to help.

About the Author

Michael Hoffman, Technical Director – nvisia

Michael Hoffman is a Principal Architect at nvisia and a Pluralsight author with over 25 years of experience designing enterprise solutions across diverse industries. A lifelong technologist and mentor, he's passionate about bridging the gap between engineering and product development.

Michael's technical expertise spans front-end and back-end architecture, with a focus on service-based APIs and Java solutions leveraging the Spring Framework. He has implemented large-scale features on IBM platforms, including WebSphere Commerce and WebSphere Portal, and regularly works with frameworks such as AngularJS, jQuery, Apache Camel, and Java.

At nvisia, Michael helps enterprise teams navigate the transition to AI-assisted development—combining deep architectural rigor with a forward-looking approach to modern engineering practices.

Connect with Michael on LinkedIn or explore his Pluralsight courses.

Resources and Further Reading

- Anthropic - **"Effective context engineering for AI agents"** by Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield - https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
- GitHub - **nvisia SDD GloboBank RAG Governance Solution and Knowledge Base** - https://github.com/NVISIA/sdd-globobank-rag-governance-solution
- GitHub - **nvisia SDD GloboBank Fraud Detection Platform** - https://github.com/NVISIA/sdd-globobank-fraud-detection-platform
- GitHub - **nvisia SDD GloboBank Fraud Management UI** - https://github.com/NVISIA/sdd-globobank-fraud-management-ui
- GitHub - **nvisia SDD GloboBank Fraud Service** - https://github.com/NVISIA/sdd-globobank-fraud-service
- GitHub - **GitHub Spec Kit** - https://github.com/github/spec-kit
- GitHub - **"Spec-driven development with AI: Get started with a new open source toolkit"** by Den Delimarsky - https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/
- Kiro - https://kiro.dev/
- MartinFowler.com - **"Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl"** by Birgitta Bockeler - https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html
- Tessl - https://tessl.io/
- YouTube - **"The New Code"** by Sean Grove, OpenAI - https://www.youtube.com/watch?v=8rABwKRsec4

nvisia
connect. build. enable.

Contact Us

nvisia.com

info@nvisia.com

312-985-8100