Beyond vibe coding

Why spec-driven development is
the future of AI software delivery

Executive summary

The next stage of AI-assisted software delivery is not simply faster code generation, but governed, specification-driven development (SDD). While prompt engineering and vibe coding accelerate experimentation and prototyping, they remain insufficient for enterprise-scale software production because they rely on fragmented context, human steering and after-the-fact correction.

We propose an implementation contract that combines two structured knowledge sources: a functional knowledge graph defining business capabilities, data semantics, process flows, rules, acceptance criteria and test data together with a technical knowledge graph defining enterprise-approved architecture, patterns, standards, security controls, deployment models and operational constraints.

Although these graphs may become large in enterprise settings, they can be narrowed through 'epic-level' functional sub-graphs and task-aware technical constraint selection, ensuring that AI receives only the bounded context relevant to the work at hand.

Importantly, these specifications are machine-actionable while remaining fully human-readable, verifiable and versionable. Together, the knowledge graphs constrain the AI's solution space, reduce ambiguity, improve traceability and establish the foundation for AI software foundries: repeatable, governed delivery systems in which agents can design, code, test, document, secure and validate software with greater control, compliance and token efficiency.

Introduction

In an era where generative AI can already write, refactor, test and explain code, the real enterprise challenge is no longer code creation. It is control. Enterprises do not simply need more code, faster. They need software that is functionally correct, but also secure, maintainable and compliant with enterprise reference architectures and engineering standards.

This is where much of today's AI software delivery falls short. Prompt engineering helps developers ask better questions. Vibe coding helps teams move quickly from idea to prototype. Both are valuable, but they rely heavily on human steering, local context and after-the-fact review. While they are powerful for individuals, they are not sufficient to build a repeatable enterprise software production system.

The next evolutionary step in software delivery automation is specification-driven development (SDD): giving AI a structured and detailed understanding of what the software must do before asking it to generate code [1, 2, 3].

This structured approach has important implications for enterprise-level token usage efficiency, where excessive token consumption can translate directly into substantial LLM costs [4, 5]. By replacing open-ended exploratory prompting with specification-driven execution, SDD reduces repeated clarification, assumption correction and iterative code regeneration of the same code. Capturing requirements, interfaces, acceptance criteria, data contracts and tests upfront precisely narrows the model's solution space, limits hallucinated decisions and avoids restating business rules across prompts. The consequence is fewer iteration loops that directly result in lower token consumption as well as faster generation of production-grade code.

Note that the specification itself is fully agnostic as to the foundational/open source LLM model that is used, the agentic framework that is employed and underlying tooling that forms part of the DevOps pipeline. Hence, the concepts outlined here have broad applicability.

The following sections offer a deeper dive into what an enterprise-grade specification should look like.

SDD: on the functional specification

Every application exists to perform one or more business functions. It processes data, follows business rules, executes process steps and must be tested with known scenarios and test data. Traditionally, these requirements live across documents, spreadsheets, user stories, code comments, test packs and the memories of experienced engineers.

AI can consume these artifacts, but disconnected information leads to disconnected reasoning. Disconnected reasoning requires AI to discover the connections through inference, which in excess leads to an increase in hallucinations and ultimately leads to wrong answers and rework. To generate software reliably, AI needs more than documents. It needs a map.

That map is a functional knowledge graph. It connects business capabilities, data definitions, process flows, business rules, epics, test cases and test data into a single traceable structure. Instead of asking AI to infer intent from fragments, the graph tells AI what matters and how the parts inter-relate.

A practical implementation example of such a map is a seven-dimensional functional specification graph linking business taxonomy, data taxonomy, processes, rules, epics, test cases and test data into a cohesive requirements model. The dimensions of such a functional specification and how they interlock are shown in Figure 1.

Figure 1:
Candidate functional ontology and the interlock among the seven dimensions

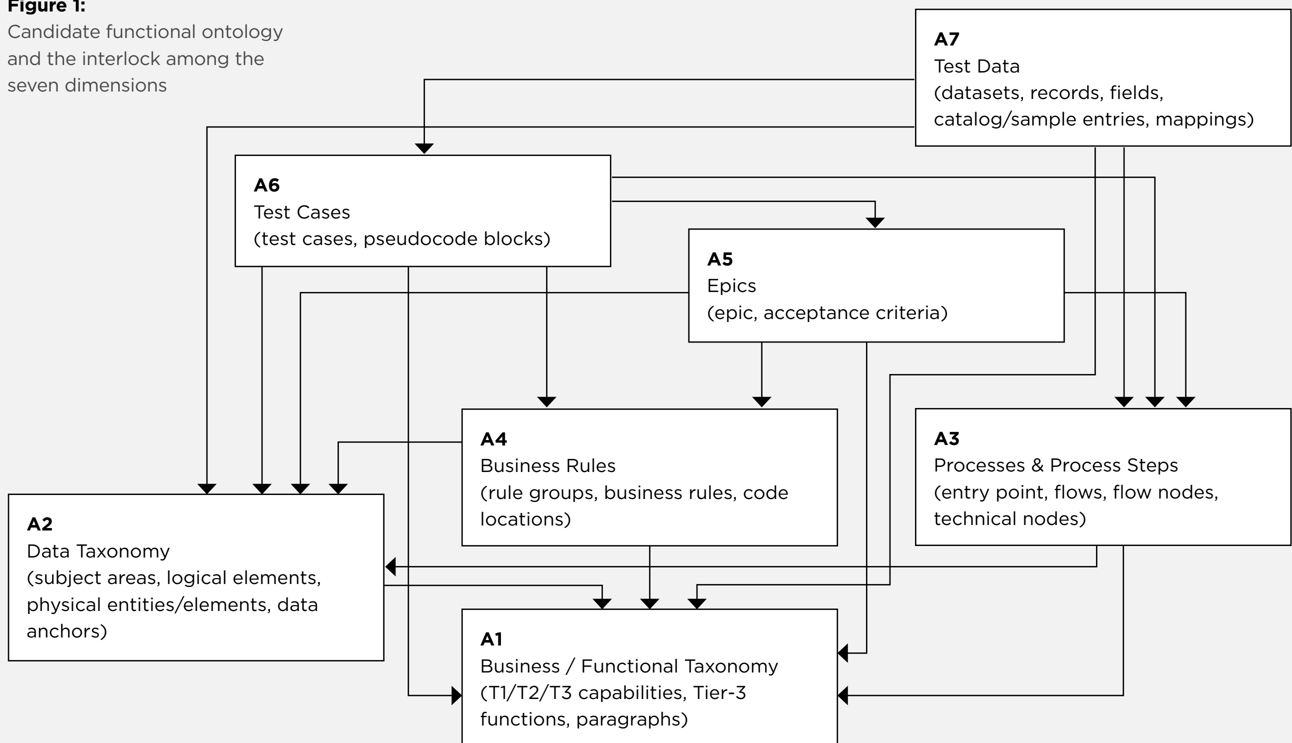


Figure 2 demonstrates what such a specification looks like in practice for a small banking mobile application.

Figure 2: An example of what the seven-dimensional functional graph looks like for a small mobile banking application

Note that the input source for the functional graph may be functional requirements in the case of greenfield software development or legacy source code in the case of a modernization project. Also note that the functional knowledge graph can be queried and divided into a collection of sub-graphs to enable iterative development across the code base via the epics node construct. This ability to slice the functional graph into smaller, context-relevant segments while preserving traceability and semantic completeness is essential when the functional specification becomes too large to fit easily within the LLM's context window.

A graph-based functional specification as described here represents a major leap forward by allowing AI to understand what business behavior must be preserved, which data elements are involved, which processes to build, which rules must be enforced, which tests prove success and – most importantly – how all these constructs interlock as given fact, as opposed to details that need to be discovered through inference.

However, production-ready enterprise software is not only judged by what it does. It is also judged by how it is built.

SDD's missing half: technical constraints

A functionally correct solution can still be wrong for the enterprise. It may use the wrong architecture patterns, the wrong domain decomposition, the wrong runtime, the wrong integration patterns, the wrong security model or the wrong deployment approach. It may work, yet violate the enterprise's reference architecture, engineering standards, coding practices, observability requirements or operational controls.

This is why AI software generation requires a second knowledge source: a technical constraint document, which we also choose to represent as a knowledge graph.

This second technical (constraint) graph is derived from the enterprise's technical rules of software construction. For example, it captures approved cloud patterns, coding standards, security controls, integration patterns, data architecture standards, test gates, CI/CD requirements, infrastructure templates, resilience policies and so on.

Similar to the functional graph, dimensions within the technical graph are wired up to other dimensions to reduce AI inference pertaining interconnected aspects of the technical constraint set.

A logical view of what such a knowledge graph might look like is depicted in Figure 3.

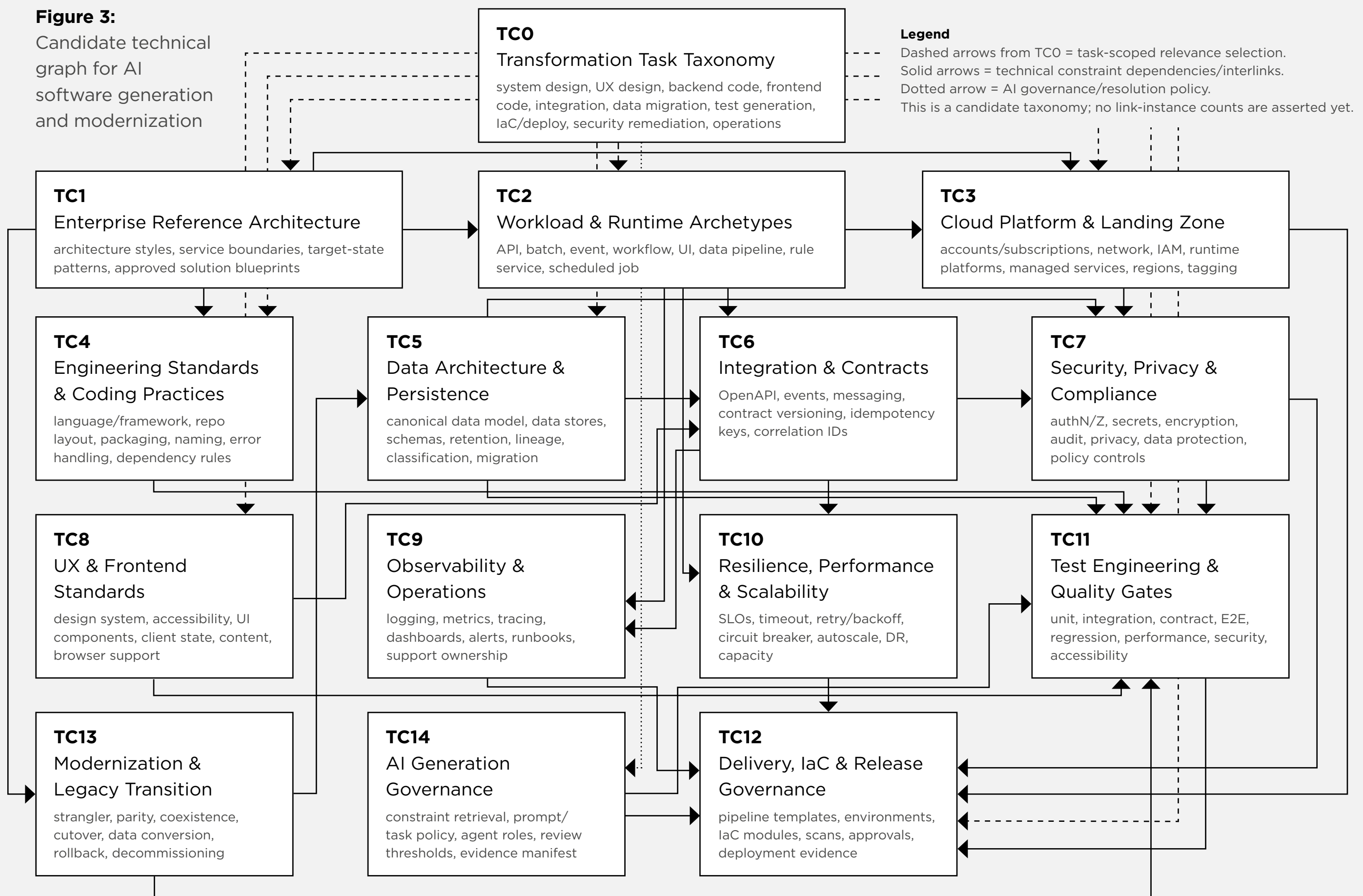
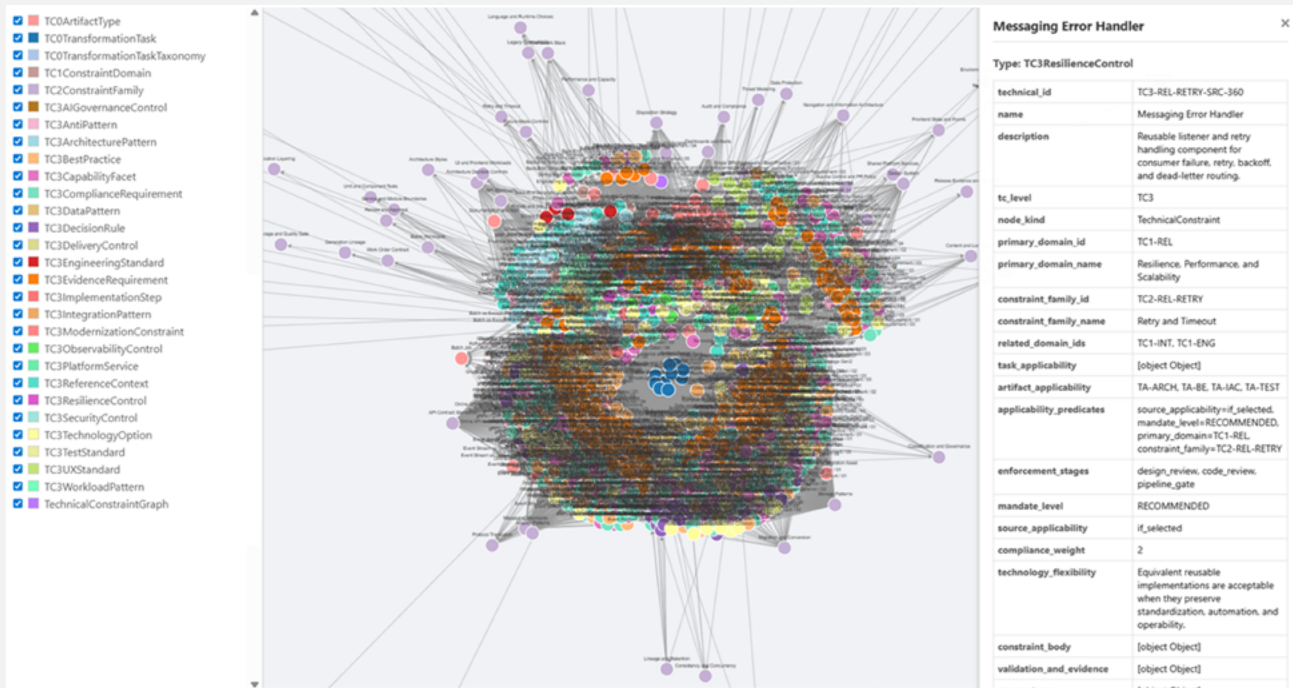


Figure 3: Candidate technical graph for AI software generation and modernization

An example of what a practical implementation of this ontology looks like is shown in Figure 4.

Figure 4: Candidate technical graph for AI software generation and modernization using the ontology described in Figure 3



Technical constraints must be task-aware

Importantly, however, not every technical constraint applies to every task that AI performs while generating the design artifacts, code and test cases. For example, a system design task needs architecture patterns, service boundaries, data ownership, integration models and resilience expectations. A backend coding task needs architecture and framework standards, data access rules, API contracts, error handling, logging, testing and security controls. A front-end task needs design system standards, accessibility, client-side validation, browser support and API consumption rules. An infrastructure task needs landing zone, network, identity and access management (IAM), ‘secrets’, tagging, deployment, monitoring and policy-as-code constraints.

This means the technical graph must be task aware. It should help AI select the constraints that are relevant to the work being performed at any given moment in time. Without this, AI will either be under-constrained and produce non-compliant software or over-constrained and become inefficient.

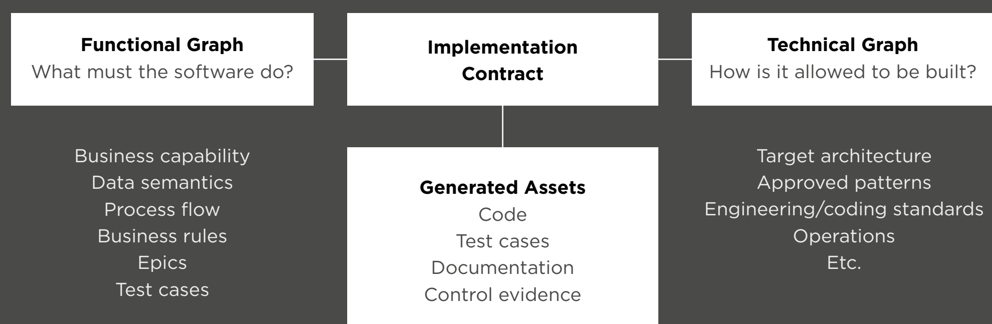
The goal is thus not to give AI every constraint all the time. The goal is to give AI the right constraints at the right moment. In practice, this is illustrated in Figure 3 through the top node: TC0 - Transformation Task Taxonomy. This node decides which other nodes need to be activated, depending on the task (design, back-end coding, unit test generation, front-end coding, etc.) the LLM is performing at that point in time.

In this we also address a practical scaling concern: both the functional and technical graphs may become very large in enterprise environments. However, the functional graph can be narrowed through epic-level sub-graph slicing, while the technical graph can be filtered by the specific software development lifecycle (SDLC) task being performed. As a result, AI does not need the full enterprise context at all times; it receives the right - and much smaller - bounded context for the work at hand.

Conclusion: from AI coding to an AI software foundry

When the functional graph and technical graph are joined, they create an implementation contract and remove ambiguity. This is shown in Figure 5. AI is no longer free to choose any plausible design. It is constrained to generate software inside the enterprise-approved solution space and verified against business intent.

Figure 5:
The implementation contract defines both what AI must build and how it must build it within enterprise-approved constraints



The implementation contract shown in Figure 5 tells AI the required business behavior, the relevant data, the process flow, the rules, the tests and the test data. It also tells AI the approved architecture, platform, runtime, coding practices, security controls, observability patterns, deployment model and quality gates.

This is fundamentally different from a prompt. A prompt asks for an output. An implementation contract defines an obligation. Furthermore, although optimized for machine execution, these specifications remain fully human-readable, verifiable and versionable. This makes them suitable not only for AI generation, but also for review, governance, auditability and controlled evolution over time.

This is the foundation of an AI software foundry: a repeatable production system where specialized AI agents can design, code, test, document, secure, deploy and validate software using governed specifications and constraints.

As enterprises move beyond experimentation, the objective should not be more AI-generated code. It should be faster, safer and more explainable software change.

Prompt engineering gave us better conversations with AI. Vibe coding gave us faster prototypes. Functional specifications gave us business-correct generation. An implementation contract that uses dual knowledge graphs gives us enterprise-compliant AI software foundries.

Moreover, this shifts the economics of AI delivery: tokens are no longer consumed repeatedly re-establishing context, correcting assumptions or regenerating work. The implementation contract makes context reusable, bounded and machine-actionable, allowing token spend to be directed toward building and validating software correctly from the outset, rather than inefficiently rediscovering requirements that should have been defined upfront.

References

1. https://github.com/github/spec-kit
2. [2602.00180] Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants
3. Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl
4. Uber Burns Its 2026 AI Budget In Four Months On Claude Code
5. Microsoft retreats from Claude Code as AI costs soar

Authors

Gerhardt Scriven, Executive Director, **gerhardt.scriven@capco.com**

About Capco

Capco, a Wipro company, is a global technology and management consultancy shaping change in the financial services and energy industries. For almost 30 years, we have been trusted to help our clients adapt, transform and create long-term value across capital markets, banking, payments, insurance, wealth and asset management, and the energy and utilities sectors. Our deep industry expertise, partnership mindset and award-winning Be Yourself At Work culture are amplified by our strengths in advisory, technology, data and AI innovation. We support our clients to establish clear priorities, connect vision to value, and deliver measurable impact when the stakes are highest.

Expert-led, AI-infused, impact-focused – we do not just respond to change, we help shape it. To learn more, visit **www.capco.com** or follow us on LinkedIn, Instagram, Facebook, and YouTube.

Worldwide Offices

APAC
Bengaluru – Electronic City
Bengaluru – Sarjapur Road
Bangkok
Chennai
Gurugram
Hong Kong
Hyderabad
Kuala Lumpur
Mumbai
Pune
Singapore

Middle East
Dubai

Europe
Berlin
Bratislava
Brussels
Edinburgh
Frankfurt
Geneva
Glasgow
London
Milan
Munich
Paris
Vienna
Warsaw
Zurich

North America
Charlotte
Chicago
Dallas
Houston
New York
Orlando
Toronto

South America
Rio de Janeiro
São Paulo

capco.com

CAPCO
a **wipro** company

© 2026 The Capital Markets Company (UK) Limited. All rights reserved.

JN_0000