Project Management for Novel Work

A historical account of the methodologies that tried to manage software and systems that had not been built before, the limitations each revealed in use, and the practices that have survived seventy years of corrections

Murray Cantor PhD

Executive Summary

This paper concerns project management for novel software and systems. Its argument rests on a short chain of reasoning: novel work proceeds with incomplete information; incomplete information is uncertainty; uncertainty, together with the consequences of being wrong, is risk. The central activity of managing novel work is therefore not the execution of a fixed plan but the systematic reduction of uncertainty through learning. A methodology is well-formed to the extent that it structures, accelerates, and preserves that learning; ill-formed to the extent that it treats early decisions as commitments rather than as hypotheses.

The history of project management is largely the history of the field's repeated rediscovery of this principle. It spans seventy years and forms a sequence of methodologies, each with a fundamental limitation in its governing assumptions, each exposed when extended beyond its proper regime, and each corrected by the next. This paper traces the lineage from IBM's early phased project control systems through Waterfall, the systems-engineering V-Model, the Spiral Model, the Rational Unified Process, Extreme Programming, the Software Factory movement, Lean, and contemporary Agile frameworks, to the Capability-Based Planning (CBP) approach now codified in U.S. defense and aerospace program management.

For each methodology, the paper outlines the motivation for its adoption, the limitations the industry discovered in practice, and the practices that endured. The pattern is consistent: each framework managed the execution of a plan more effectively than the risk that the plan itself was wrong. None answered the question that persisted across the lineage — how is the validity of the plan itself to be analyzed? CBP, as adopted by the U.S. Department of Defense and NASA, is one answer—not the only viable answer, but the one most visible in current institutional practice. It organizes planning around the outcomes a program is committed to delivering and the capabilities required to produce them, rather than around the tasks believed to produce those outcomes and treats the maturation of those capabilities as the principal artifact of the program. Both risk management — the identification and tracking of named risk events — and risk analysis — the quantitative propagation of uncertainty into a posterior probability of success — become native to the plan rather than appended to it. Where CBP is paired with the digital engineering and digital twin infrastructure now mandated by DoD Instruction 5000.97 and increasingly common in NASA and commercial aerospace practice, the pairing addresses the shortfalls the lineage exposed. The capability model expresses what must be true for the program to succeed; the digital twin makes the underlying assumptions continuously testable in software rather than only at discrete physical milestones. Learning and uncertainty reduction, the principle the lineage spent seventy years discovering, becomes the structure of the plan itself, and the twin becomes the substrate on which that learning runs at program scale. Other approaches exist and continue to evolve; this paper describes the one currently codified at institutional scale and treats it as a useful reference rather than as the final word.

1. Introduction

The argument of this paper rests on a short chain of reasoning. Novel work — the development of something that has not been built before — proceeds with incomplete information about what is being built, how to build it, what it will cost, how long it will take, and whether it will produce the intended

outcome. Incomplete information is uncertainty. Uncertainty, combined with the consequences of being wrong, is risk. The central principle of project management for novel work is therefore not the execution of a fixed plan but the systematic reduction of uncertainty through learning. A methodology is well-formed to the extent that it structures, accelerates, and preserves learning; ill-formed to the extent that it treats early decisions as commitments rather than as hypotheses. This single principle is the lens through which the lineage that follows can be read.

Every project management methodology is a theory for handling uncertainty. Early methodologies assumed uncertainty could be eliminated through planning. Later ones accepted uncertainty as inevitable and sought to absorb it through iteration. The most recent generation accepts that even the goals may shift during execution. The lineage reflects the field's progressive discovery that the principle above — learning and uncertainty reduction as the central activity — matters more than the methodology in which it is embedded.

The most visible current attempt to address the shortfalls the lineage exposed pairs Capability-Based Planning with the digital engineering and digital twin infrastructure now codified in DoD Instruction 5000.97 and increasingly common in NASA and commercial aerospace practice. CBP supplies the plan structure — outcomes, capabilities, and the assumptions on which each depends — while the digital twin provides the substrate for continuously testing those assumptions in software. The pairing is an approach, not the approach; other responses exist and continue to evolve. This paper examines the response most clearly codified at the institutional scale today.

Understanding why each method was created and why each was superseded is a useful frame for the practice as it stands today. The historical record suggests methodologies have rarely failed because the method was inherently defective; they have more often failed because the method was extended to a class of risk it was never designed to handle, or because its structure obstructed learning rather than enabling it.

This paper concerns project management for novel work — the development of software and systems never built before, where the specification is, in significant part, the work itself, and where uncertainty about outcomes, requirements, performance, cost, and duration is intrinsic rather than a defect to engineer out. This is a different regime from well-understood, repeatable work — maintenance of a deployed system, manufacturing to a settled design, porting between known platforms — for which a different family of methods works well. The two regimes are often conflated; the consequences of applying the wrong family to the wrong regime are a recurring theme in the record.

The lineage developed largely within software because that was where documented experience with novel work accumulated most rapidly through the late twentieth century. Each generation of software developers confronted artifacts never built before — first compilers, then operating systems, then distributed systems, then web-scale platforms, then machine-learning systems — and each generation discovered, often the hard way, that techniques developed for the previous generation's well-understood work did not transfer. The methodologies named in this paper — Waterfall, the V-Model, Spiral, RUP, XP, the Software Factory movement, Lean, and Agile — originated as software movements, and what survives of each is the part that addressed intrinsic novelty rather than the part that tried to manage it away. The pattern is not specific to software. The same dynamic appears wherever novel systems are developed under uncertainty: defense acquisition, spacecraft and aerospace programs, pharmaceutical research, infrastructure megaprojects, large-scale organizational transformation. Capability-Based Planning was not developed in software at all; its origins are in defense planning, and its institutional adoption at the U.S. Department of Defense and NASA is in domains where the artifact is a system, not software. Software is one instance of the broader category of novel work, not the category itself.

2. The Lineage of Project Management Structures

2.1 The IBM Phased Project Lifecycle (1960s)

Motivation

Through the 1960s, IBM and its peers faced unprecedented challenges in managing software and hardware development at a scale without precedent. The OS/360 program — then the largest software effort ever undertaken — exposed the impossibility of running large engineering efforts on intuition alone. IBM responded by formalizing a phased lifecycle — requirements, design, implementation, testing, deployment — each gated by formal review and documentation. The motivation was control: executives needed predictability, customers needed contracts, and engineers needed a shared structure.

Weaknesses

The phased lifecycle assumed that requirements could be fully and correctly captured at the outset. In practice, requirements changed, designs proved infeasible, and integration revealed errors that should have surfaced earlier. A requirements defect found during testing required rework across every downstream phase. The model treated learning as a defect rather than a feature of complex work.

2.2 The Waterfall Model (1970)

Motivation

Winston Royce's 1970 paper formalized the sequential lifecycle the industry had been practicing informally. Ironically, Royce himself warned that a strictly sequential approach was "risky and invites failure" and proposed feedback loops between phases. The defense and government procurement community adopted the sequential portion of his model and codified it into standards such as DoD-STD-2167. The motivation was contractual clarity and the auditability of milestones.

Weaknesses

Waterfall's first weakness is structural: it stovepipes the disciplines. Requirements analysts hand off to designers, designers to coders, and coders to testers. Each discipline works behind a wall, with its output frozen before the next begins. Knowledge that surfaces in a later phase — a misunderstood requirement exposed during coding, a design flaw exposed during integration — cannot easily flow back upstream because the upstream phase is already closed.

The second weakness follows directly: no mechanism exists to adjust the plan when the situation changes. Waterfall is the project-management equivalent of a ballistic missile. Once launched, the trajectory is fixed and cannot track a moving target. If requirements shift, the threat evolves, or the original need turns out to have been misunderstood, the plan continues on its original arc and delivers to a point the target no longer occupies. Royce recognized this, which is why he proposed feedback loops between phases — guidance the procurement community largely ignored.

The third weakness is back-loaded risk. Because validation occurs only at the end of the lifecycle, late-discovered defects are exponentially more expensive to fix than those found early. The project appears healthy through most of its lifecycle, then collapses during integration and acceptance. Waterfall manages schedule and scope; it does not manage discovery, change, or learning.

2.3 The Systems Engineering V-Model (Forsberg and Mooz, 1991)

Motivation

While Waterfall dominated software practice through the 1980s, the systems engineering community pursued its own response to the same problems. Large defense and aerospace programs were building systems composed of hardware, software, and human operators, and the sequential lifecycle fit poorly.

In 1991, Kevin Forsberg and Harold Mooz presented their V-Model at the first annual conference of the National Council on Systems Engineering, later INCOSE. The model is now the iconic illustration of systems engineering and the dominant teaching framework in the field. It is in active use across defense acquisition, aerospace, automotive, and medical-device development, and is referenced in national standards, including DoD 5000-series acquisition policy, ISO/IEC/IEEE 15288, and the NASA Systems Engineering Handbook.

The V's left leg descends through successive levels of decomposition — concept of operations, system requirements, system architecture, subsystem design, component design, and implementation. The right leg ascends through corresponding levels of integration and verification, each validating against the artifact produced at the same horizontal level on the left: components against their design, subsystems against the architecture, the system against its requirements, and the delivered product against the original concept of operations. Forsberg and Mooz intended the V as an advance over Waterfall, making verification a first-class citizen of the lifecycle rather than an afterthought. They acknowledged Boehm's Spiral but argued that it obscured the systems engineer's role, whereas the V made it clear.

Weaknesses

Structurally, the V is a Waterfall with a verification mirror. The left leg inherits Waterfall's sequential decomposition — requirements before architecture, architecture before design, design before implementation — with each level frozen as a baseline before the next begins. The same stovepiping reappears, now compounded across system-hierarchy levels. A misunderstanding at the requirements level propagates down through subsystem and component requirements and into implementation and is detected — if at all — only on the right leg, where correction is most expensive. The V makes verification visible; it does not make the discoveries that drive verification arrive earlier.

The deeper weakness is the verification mirror itself. Each right-leg activity validates the deliverable against the baseline set at the corresponding level on the left. If the baseline was correct, verification confirms a sound product. If the baseline was wrong, verification confirms a faithful implementation of the wrong product. The V succeeds at verifying the wrong thing. The canonical case is the 1999 loss of the Mars Climate Orbiter: every level of the V was completed and signed off, yet the spacecraft was destroyed because a navigation interface between subsystems used inconsistent units — pound-force-seconds on one side, newton-seconds on the other. The defect was in the agreed interface specification, faithfully implemented on both sides. Every verification activity passed because each checked against the flawed baseline rather than the underlying mission outcome. The V offers no structural mechanism for detecting that a baseline is wrong, because verification operates within the frame the baseline established. Section 4.5 returns to the MCO case to describe how a digital twin running the actual subsystem outputs through a flight-dynamics model — verifying the system against the physics rather than against the specification — would have surfaced the discrepancy long before launch.

Like Waterfall, the canonical V offers no provision for course correction during execution. The left-leg phases are sequential and baselined; changing a higher-level requirement after lower-level design has begun is costly and discouraged. The right-leg phases are ordered by build order and cannot begin in earnest until the corresponding components exist. The classical V, like Waterfall, is a ballistic trajectory plotted at launch — better instrumented, better illustrated, more disciplined in verification, but still aimed once and committed.

In practice, the canonical V is rarely run today. Since the V was published, practitioners have produced variants that soften its severity, and several are now standard in regulated industries. The incremental V-Model executes a sequence of smaller V's, each delivering a partial capability and feeding the next; it is common in defense and aerospace acquisition. The iterative V repeats V-cycles for the same capability, refining with each pass, and is used in safety-critical automotive (ISO 26262) and medical-

device development. V-Model XT, codified as a German federal standard, allows project-specific tailoring of phases, artifacts, and approvals. Hybrid V-plus-Agile approaches drive team-level iteration within the V's systems-engineering structure and are increasingly common where regulators require V&V rigor and the program also needs delivery cadence. Modern digital-engineering practice in DoD acquisition treats left-leg artifacts as living models — continuously updated SysML, evolving interface specifications, executable architectures — with right-leg verification running in parallel against the latest baseline rather than waiting for build completion. Each variant sequentially reduces the V and shortens the feedback loop between left-leg decisions and right-leg evidence.

These variants soften Waterfall, and meaningfully so. What they do not address is the deeper weakness the canonical V exposed: the right leg verifies against the baseline established on the left, and a shorter feedback loop only means shorter cycles of confirming the wrong baseline when the baseline is wrong. An incremental V verifies each increment against its slice of the baseline. An iterative V verifies the same baseline more times. A continuously integrated V verifies the latest baseline against the latest evidence. None structurally addresses the case in which the baseline itself is the defect. The Mars Climate Orbiter unit-convention defect lived in an interface specification that both sides faithfully implemented; an incremental V would have verified the same defective interface in each increment, an iterative V on each pass, and a continuously integrated V on every commit. The variants improve the V's tempo; they do not change what the V verifies against.

The V — canonical or variant — remains in active use, and many programs run successfully under it. The accumulated experience is that the V works well when the system closely resembles those the organization has built before — when requirements and architecture can be specified with high confidence at the outset — and works poorly in genuine novelty, where the validity of those early specifications is precisely what is at stake. Recent INCOSE writing acknowledges that the V's interpretation has been challenged by systems-of-systems development, value-stream thinking, and the extension of systems engineering into commercial sectors where its assumptions often do not hold.

2.4 The Spiral Model (Boehm, 1988)

Motivation

Barry Boehm proposed the Spiral Model as a risk-driven alternative to the Waterfall. Each spiral consisted of objective-setting, risk analysis, development, and planning for the next iteration. The innovation was placing risk analysis at the center of the methodology. Instead of advancing through phases on a fixed schedule, the project advanced only after the risks of the next iteration had been identified and mitigated.

Weaknesses

Spiral's principal operational weakness was the lack of a clear definition of done. The model told the team how to advance from one iteration to the next — assess risk, build a prototype, plan the next spiral — but did not tell the team when to stop. Each spiral surfaced new risks, and addressing them generated the next spiral, which in turn surfaced further risks. In principle, this was the point: the project advanced only as fast as risk was retired. In practice, it meant projects could spiral indefinitely, consuming budget and schedule without converging on a deliverable. A method designed to control risk became, in many organizations, a method that could not control itself.

Supporting weaknesses compounded this. Each iteration required formal risk analysis, prototyping, and stakeholder review — overhead that relied on risk-analysis expertise most teams lacked. Combined with the absence of a completion criterion, the model proved easier to start than to finish. Spiral remained influential as a way of thinking about risk-driven development but was largely abandoned as an operating model because projects spiraled out of control rather than into completion.

2.5 The Rational Unified Process (1990s)

Motivation

The Rational Unified Process (RUP), developed at Rational Software and later acquired by IBM, sought to combine the discipline of phased delivery with the iteration of the Spiral model. RUP's most durable contribution was its four-phase model — Inception, Elaboration, Construction, and Transition — which it explicitly framed not as sequential work buckets but as states of increasing maturity of understanding about the system being built. Inception established that there was a problem worth solving and a rough sense of scope. Elaboration mitigated the major architectural and requirements risks, producing an executable architecture and a credible plan. Construction scaled up production once the principal uncertainties had been resolved. Transition handed the system over to its users. The phases were anchored not to dates on a schedule but to the achievement of specific risk-reduction milestones, and each phase was executed through multiple iterations. RUP also introduced the use case as a unit of requirements and tightly integrated it with the Unified Modeling Language.

Weaknesses

The phase model was not the problem. The maturity-of-understanding framing — Inception establishes the problem, Elaboration retires the principal risks, Construction scales up production, and Transition delivers — remains one of the clearest accounts of how complex work actually progresses, and it survives in much current thinking about staged investment under uncertainty. The problem was everything wrapped around the phases.

RUP was prescriptive and artifact heavy. The framework defined dozens of roles, nine disciplines, and more than sixty artifacts, each with templates. In principle, RUP was meant to be tailored — adopters were to select only the artifacts and roles their project required. In practice, the sheer volume of named deliverables, combined with the difficulty of justifying any omission, pushed organizations to adopt the framework wholesale. RUP implementations frequently degenerated into ceremony: documentation produced to satisfy the process rather than to inform the work, and templates filled out to demonstrate compliance with a methodology rather than to advance understanding of a system.

The phase model and the artifact catalog were tightly coupled. Each phase had expected deliverables, defined at a level of detail more appropriate for a multi-year defense or enterprise program than for the broad range of work RUP was marketed to address. The right idea — work passing through stages of increasing understanding — was yoked to the wrong delivery model: a heavyweight, documentation-centric apparatus that the industry would soon push back against, first through XP and then through Agile.

2.6 Extreme Programming (late 1990s)

Motivation

Extreme Programming (XP) emerged as a deliberate rejection of heavyweight processes. Kent Beck and his collaborators argued that the practices known to produce good software — testing, code review, refactoring, and close customer contact — should be applied continuously rather than at gated milestones. XP introduced pair programming, test-driven development, continuous integration, and the on-site customer. The motivation was to shorten the feedback loop between writing code and learning whether it was correct.

Weaknesses

XP's engineering practices were a genuine advance and have outlasted XP itself. Test-driven development, continuous integration, refactoring, pair programming, and collective code ownership survived XP's decline as a methodology and are now baseline practices across the industry. The

discussion that follows is therefore not about those practices but about the methodology that introduced them.

XP is a methodology for the inner loop. It optimized the seconds-to-minutes feedback cycle between writing code and learning whether it worked, as well as the days-to-weeks cycle between the team and the on-site customer. It said nothing — and was designed to say nothing — about the months-to-years cycle that governs business planning: which capabilities the enterprise needs, which investments compete for the same dollars, how a portfolio of work is sequenced against strategic outcomes, and how the risk that any of those decisions is wrong should be analyzed.

XP's reliance on a single empowered on-site customer made this gap structural. The method assumed the customer possessed sufficiently stable knowledge of the desired behavior and that intent could be made sufficiently explicit through continuous interaction. There was no provision for the case in which the customer did not yet know, or for the case in which the customer's stated needs did not align with the broader enterprise's outcomes. XP excelled at building the thing right; it offered no structure for deciding which things were worth building.

2.7 The Software Factory Movement (1970s–1990s)

Motivation

The term "software factory" was coined in the late 1960s and seriously pursued from the 1970s into the early 1990s, most visibly by Japanese firms — Hitachi, NEC, Toshiba, Fujitsu — and in parallel by U.S. efforts, including IBM's process programs and the work of the Software Engineering Institute. The premise was that software development should be industrialized: standardized processes, codified methods, reusable components, captured metrics, and defined skill levels would together turn programming from a craft into a production discipline. The motivation was the same that has driven every methodology in this lineage — predictability of cost, schedule, and quality at scale — pursued by analogy to the most successful production systems of the twentieth century.

Weaknesses

The software factory rested on a category error that would recur, in different packaging, for the next thirty years. Manufacturing is a deterministic activity: the artifact is fully specified before production begins, the same artifact is produced repeatedly, and the process suppresses variation around a known design. Quality control measures deviation from that standard. In that setting, variation is waste, and eliminating waste is almost synonymous with improvement.

Novel system development is a fundamentally different kind of activity, whether the artifact is software, a spacecraft, a weapons platform, a new pharmaceutical, or a regulatory regime. The artifact is not specified in advance — its specification is, in significant part, the work. Two genuinely new development efforts are rarely the same in any deep sense; if they were, the second would be a copy rather than a development project. The process is not to suppress variation but to manage discovery under uncertainty, and variation in approach is often the path to a better solution rather than a defect to engineer out. Treating discovery-oriented development as factory output applies the wrong mathematics: it imposes a model of repeatable production on work whose central problem is not repetition. The factory model fits parts of any system's lifecycle — maintenance, porting, product-line engineering of well-understood variants, sustainment of fielded systems, and manufacturing of physical units against a settled design — precisely the cases where its assumptions hold.

The software factory movement faded by the mid-1990s. The factories that produced impressive metrics tended to be those whose work was, in fact, repetitive — maintenance, porting, and well-understood business applications. Where work was genuinely novel, the factory metaphor obscured rather than illuminated the problem, and methods designed for repeatable production drove perverse behavior when applied to discovery.

2.8 Lean Software Development (2000s)

Motivation

Lean originated in Toyota's manufacturing system and was adapted to software and project work by Mary and Tom Poppendieck. Its core insight was that most time, effort, and cost in any process are consumed by waste — work that does not produce value for the customer. Lean introduced principles such as eliminating waste, amplifying learning, deferring commitment, delivering fast, and optimizing the whole. The motivation was efficiency, but the deeper aim was to make the flow of value visible and measurable.

Weaknesses and Proper Domain

Lean's limitation in the lineage is the same one that the Software Factory exposed, in lighter dress. Where the factory borrowed from mass production, Lean borrowed from the Toyota Production System — a more sophisticated manufacturing tradition, but one, nonetheless. The underlying assumption is the same: that the dominant problem is the efficient flow of value through a known process. Applied to novel development, where value is not yet defined, and the process is precisely what the work is discovering, production-line metrics — throughput, cycle time, defect rate per unit — drive perverse behavior. Eliminating "waste" eliminates the exploratory effort that was the point. Optimizing flow optimizes the team's ability to deliver the wrong thing faster. Lean offers no language for the dominant risk in novel work: that the team's current understanding of the problem is itself wrong.

The proper domain for Lean's methods is the regime where its assumptions hold: the change-request process for a deployed product. Once a system is in operation, the stream of incoming changes — bug fixes, feature requests, performance improvements, regulatory updates, sustainment items — is a flow of reasonably well-understood units within a reasonably stable process. Each change has a definable scope and moves through intake, triage, design, implementation, review, testing, and release. Cycle time is measurable and meaningful. Work-in-progress can be limited effectively. Batch sizes can be tuned, queue lengths controlled, and cost of delay computed. The production-flow economics Reinertsen formalized — and those SAFe and other scaling frameworks later borrowed — apply here, because the regime they assume is the one that actually exists.

Lean's contribution is therefore real and durable, but specific to a phase of the system's life rather than to novel development as a whole. The misapplication critiqued above occurs when Lean's methods are extended upstream into the discovery work that produces a new system. The methods themselves remain the right tools for the change-request process, the sustainment backlog, the operations queue, and the steady-state delivery pipeline against a fielded product. The lineage's lesson on Lean is the same it teaches on the Software Factory: not that the methods are wrong, but that they have a proper domain, and that the productivity of the field as a whole depends on distinguishing that domain from the discovery work for which different methods are required.

2.9 Agile Methods (2001 to present)

Motivation

The Agile Manifesto of 2001 codified a shared set of values across XP, Scrum, Crystal, and related approaches: individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.[1] The motivation was to make adaptation a first-class property of the delivery system. Scrum in particular spread rapidly, offering a lightweight framework of roles, events, and artifacts that any team could adopt.

[1]Beck, K., et al. (2001). Manifesto for Agile Software Development. https://agilemanifesto.org

Weaknesses

As with XP, Agile's core operational practices have outlasted the broader methodology. Scrum's lightweight ceremonies — short iterations, daily stand-ups, sprint reviews, and retrospectives — remain widely used to coordinate a small team and surface impediments quickly. Kanban's flow-visualization practices are similarly durable. The discussion that follows concerns the broader methodology rather than these practices.

In many organizations, Agile evolved into a delivery-centric practice that remained insufficiently connected to enterprise planning. The Manifesto's authors were developers describing the conditions under which developers do their best work. That perspective is legitimate, but it was not balanced by an equivalent account of the conditions under which the enterprise does its planning — and the enterprise has planning work to do. Executives need cost commitments to set prices and budgets. Boards need delivery commitments to allocate capital. Customers need date commitments to coordinate dependent investments. Some Agile interpretations treated these obligations as external to the delivery system rather than as constraints the system was required to meet.

The "no estimates" movement illustrates this posture sharply. Its argument was that estimates are inherently inaccurate, that estimating wastes time better spent building, and that the business should accept continuous delivery as a substitute for forecast commitments. Whatever its technical merits, the practical message was clear: the enterprise's planning obligations were not treated as obligations of the delivery system. A discipline that asks its principal stakeholder to stop requesting the information it needs to govern the organization has not solved the planning problem; it has stepped outside it. The result, in many organizations, was an Agile practice that delivered code reliably while leaving executives funding it without a credible answer to basic questions about cost, schedule, and outcomes.

The Agile community has recognized this gap and developed several metric frameworks to address it. Objectives and Key Results (OKRs), originally developed at Intel and now widely used in technology companies, link team work to measurable outcomes the team commits to deliver in a quarter. North Star Metrics anchor product teams to a single outcome measure assumed to track long-term value. Evidence-Based Management, codified by Scrum.org, frames product decisions around four value-area metrics — current value, time to market, ability to innovate, and unrealized value. The Business Agility movement extends similar instrumentation to the enterprise level. Each is a deliberate response to the observation that velocity, story points, and feature counts measure activity rather than outcomes.

These frameworks measure outcomes meaningfully. What they do not measure is capability. An outcome metric — quarterly revenue, weekly active users, time-to-market, or an OKR's key result — is a reading on the system's current state; it tells the team whether the recent past produced the result they were aiming for. A capability is the system's ability to produce a class of outcomes against a stated performance threshold, along with the assumptions on which that ability depends. Outcome metrics answer "did the last cycle work?"; capability measures answer "will the program meet its objective?" An OKR can be hit while the underlying capability erodes; a Net Promoter Score can rise while a strategic capability gap widens; a North Star Metric can trend favorably for reasons unrelated to anything the team did. Outcome metrics are necessary but not sufficient: they are not a substitute for an explicit model of the capabilities the enterprise needs and the assumptions on which each depends. OKRs measure where the system is; capabilities express what must be true for it to get where it is going .

Two further weaknesses follow. Agile excels at delivery within a team but struggles at scale; the scaling frameworks that followed — SAFe, LeSS, Nexus, Disciplined Agile — exist precisely to add the coordination and planning layers that Agile-as-team-practice declined to provide. And Agile is a methodology for executing against a backlog: silent on how the backlog is chosen, how its items relate to the outcomes the business is committed to delivering, and how the risk that the backlog is solving the

wrong problem should be analyzed. Even with outcome metrics layered on top, Agile manages the risk of building the thing wrong more effectively than it manages the risk of building the wrong thing.

SAFe's value-stream concept warrants a specific note. SAFe's economics are grounded in Donald Reinertsen's work on product development flow — cost of delay, queue length, batch size, and work-in-progress limits. That body of work has a substantial track record and is strongest in environments where work units are already well-defined and the operative question is how to flow them efficiently. Change requests against a deployed product fit that mold precisely: the product exists, the users exist, the request is a discrete unit of known scope, and the value of expediting it can be quantified. In that regime, SAFe's value streams do track value. The point at which historical experience suggests the model has been overextended is when the same approach is applied across the entire enterprise portfolio, including greenfield development and new-capability acquisition, where work units are not yet defined and the operative question is not how to flow them but whether they are the right work at all.

3. The Recurring Gap: Plan Risk vs. Execution Risk

Read together, every methodology in the lineage manages one of two risk classes well and the other poorly:

- **Execution risk** — the risk that work in progress will be late, defective, or over budget. Waterfall, the V-Model, RUP, XP, the Software Factory, Lean, and Agile all offer mature techniques for managing execution risk.
- **Plan risk** — The risk that the plan itself is wrong: that the scope, the sequence, or the assumed outcomes will not deliver the capability the enterprise actually needs. Spiral addressed this conceptually but not operationally. The others address it only incidentally.

This gap is a persistent failure mode in large, complex programs. At the enterprise scale, consequential failures often stem not from the team's inability to execute the plan, but from the plan itself, when faithfully executed, producing an outcome the enterprise no longer needed or never needed. The history shows the lineage that produced no structure treating plan risk as a central object of analysis until CBP.

3.1 A History of Discovered Limitations and Retained Practices

Read as a whole, the lineage forms a sequence of corrections rather than a menu of methods. Each methodology contained a fundamental limitation in its governing assumptions, exposed when the method was extended beyond the regime for which it was suited. Waterfall assumed requirements could be fixed in advance and that work could be stovepiped behind disciplinary handoffs; the industry discovered that complex novel systems cannot be specified in advance and that frozen handoffs prevent the learning the work depends on. The V-Model assumed that adding a verification mirror to Waterfall would address its weaknesses; the industry discovered that the V succeeded at verifying the wrong thing whenever the baseline against which it verified was itself wrong. Spiral assumed risk-driven iteration could organize a project; the industry discovered that without a definition of done, the spirals would not reliably converge. RUP assumed that a defined set of artifacts and roles could operationalize a sound phase model; the industry discovered that the artifact catalog crushed the model it was meant to support. The Software Factory and Lean assumed that software development was a production activity; the industry discovered that novel system development is a materially different kind of activity, that production-line metrics drive perverse behavior when applied to discovery work, and that the production-flow methods nonetheless have a legitimate home in the change-request process and sustainment regime where their assumptions actually hold. XP and Agile assumed that the team and its immediate customer were the right unit of organization; the industry discovered that this framing left the enterprise's planning obligations unmet and that some interpretations treated those obligations as external to the delivery system. The thread that runs through every one of these discoveries is the same:

novel work generates information as it proceeds, and any methodology that obstructs the flow of that information back into the plan — by freezing baselines, by suppressing variation, by ignoring outcome-level feedback, or by treating estimates as commitments rather than as hypotheses — will fail in proportion to the obstruction.

What survives each correction is the part of each methodology that withstands subsequent experience. RUP's maturity-of-understanding phase model survives in current thinking about staged investment under uncertainty. XP's engineering practices — test-driven development, continuous integration, refactoring, pair programming, collective code ownership — are now baseline practices across the industry. Scrum's lightweight team ceremonies — short iterations, daily stand-ups, sprint review, and retrospective — remain in widespread use as a structure for coordinating a small delivery team. Reinertsen's product-flow economics remain in use for the regime he described: discrete units of well-defined work against a deployed product. These practices are now part of the field's working vocabulary.

What did not survive was the universalizing claim each methodology made about itself. None of them proved to be an answer to the question that persisted across the entire lineage: how is the validity of the plan itself to be analyzed? Capability-Based Planning is the field's response to that question — the most fully developed one currently in institutional use. It does not discard the surviving practices; it supplies the planning and risk-analysis layer above them, the layer at which the question of whether the plan is right is now addressed in current practice.

4. Capability-Based Planning

4.1 Definition and Origins

Capability-Based Planning emerged in the late 1990s and early 2000s within defense planning communities, notably in the Australian and U.S. Departments of Defense, as a discipline for planning and controlling complex programs under deep uncertainty. CBP, as used here, should be distinguished from the superficially similar use of "capabilities" in enterprise architecture. Enterprise architecture uses capability maps to describe what an organization does, primarily as an organizing model for systems, data, and business functions. CBP, as a project planning and control structure, uses capabilities as the unit against which plans are built, options are evaluated, risk is analyzed, and progress is measured. The two practices share vocabulary but address different problems on different time horizons.

Under CBP, a capability is defined independently of any specific solution: it is what must be able to be done to achieve a required outcome, not what will be bought or built. A plan is then a structured set of decisions about which capabilities to develop, in which sequence, to which level of performance, and against which assumptions.

4.2 How CBP Differs in Practice from Earlier Methods

Practitioners working with CBP describe four structural differences between it and the methods that preceded it in the lineage:

- **Outcome anchoring.** Where Waterfall and RUP anchored planning to scope and Agile anchored it to backlog items, CBP anchors planning to the capabilities required by the outcome. Scope, schedule, and solutions become consequences of the capability set rather than its starting point.
- **Solution independence.** Because a required capability is defined without reference to a specific solution, alternative ways of delivering it can be compared on their merits — including the option of not building anything at all. This is what the historical record shows distinguishes CBP from execution-focused methods, which tended to commit early to a particular solution path.

- **Program-level coherence.** CBP operates above the individual project, coordinating multiple projects against a shared set of required capabilities. Dependencies and gaps between projects are expressed in the plan itself — a structural feature that Agile scaling frameworks address through additional layers of ceremony.
- **Risk as a first-class object.** Each capability in the plan includes an explicit statement of the outcome it enables, the assumptions underlying that outcome, and the gap between current and required performance. Risk analysis is structurally embedded in the plan rather than appended to it.

4.3 Institutional Adoption

Two of the largest and most demanding program organizations in the United States have codified capability-based approaches into their formal management policies, with instructive variation in how each has done so.

In 2003, the U.S. Department of Defense adopted capability-based requirements generation through the Joint Capabilities Integration and Development System (JCIDS), replacing the earlier service-specific Requirements Generation System. JCIDS embodied the 2001 Quadrennial Defense Review's shift from threat-based to capability-based defense planning. For more than two decades, it was the formal process for identifying, validating, and translating capability gaps into acquisition requirements across the joint force. In August 2025, the Secretary of Defense directed the disestablishment of JCIDS within 120 days.[2] The reform did not reject capability thinking; it rejected the bureaucracy that had accumulated around it. JCIDS had become notorious for taking up to 800 days to validate a single requirement document. The reform replaces that heavyweight validation with a leaner structure: a Requirements and Resourcing Alignment Board focused on a small set of Key Operational Problems, a Joint Acceleration Reserve tying priorities directly to funding, and an updated Capability Portfolio Management policy. Capabilities remain the unit of analysis; the validation apparatus has been drastically pared back. The pattern echoes RUP: the underlying model survives, while the surrounding artifact-heavy implementation does not.

NASA has codified capability portfolio management as a required input to program and project planning. NPR 8600.1, NASA Capability Portfolio Management Requirements, governs how capability portfolios are defined and managed across the Agency, and the Program Plan and Project Plan templates in NPR 7120.5F require documented concurrence on investments, divestments, and changes to capability portfolio components.[3] NASA's adoption is narrower than DoD's under JCIDS — the dominant program management framework still organizes spaceflight work by life-cycle phases and relies heavily on Joint Cost and Schedule Confidence Level (JCL) analyses — but capability portfolio management is an explicit and enforceable part of how programs are planned and reviewed.

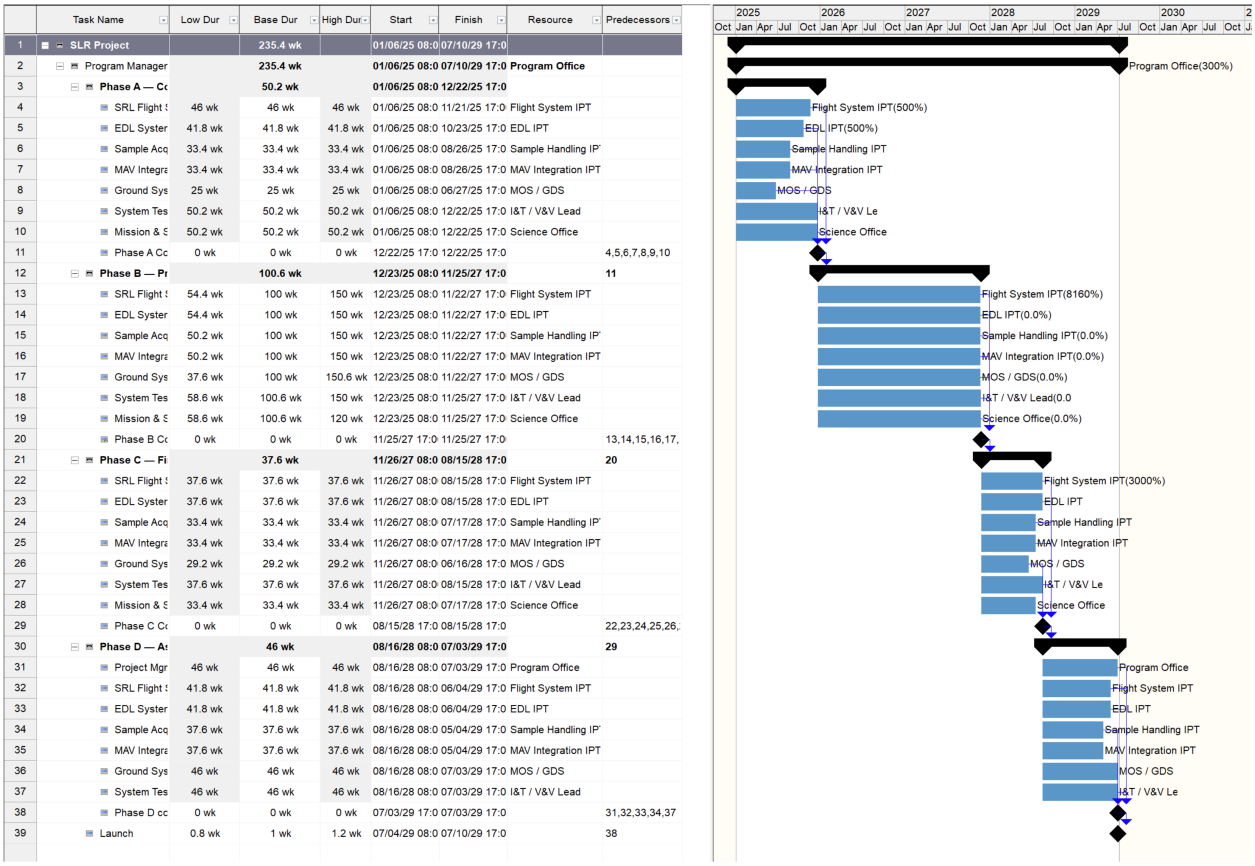
Taken together, the two institutional records show capability-based planning established at both organizations in different forms, with the DoD's twenty-two-year experience now informing a second generation of implementation. The 2025 reform retained capability framing while dismantling the validation bureaucracy around it — consistent with the broader pattern this paper traces, in which a methodology's underlying model often survives while the artifact catalog of a specific implementation does not.

4.4 Phased Without Stovepipes: A Worked Illustration

[2]U.S. Department of Defense (2025). Reforming the Joint Requirements Process to Accelerate Fielding of Warfighting Capabilities. Memorandum of the Secretary of Defense, August 20, 2025.
[3]NASA (2021). NPR 7120.5F, NASA Space Flight Program and Project Management Requirements; NPR 8600.1, NASA Capability Portfolio Management Requirements.

A common objection to capability-based planning is that any plan organized into phases must recreate the stovepiping problem that crippled Waterfall. The objection conflates two different uses of "phase." Waterfall phased work by discipline — requirements analysts in phase one, designers in phase two, coders in phase three, testers in phase four — and froze each discipline's output before the next began. CBP phases work by the maturity of understanding, the insight RUP contributed, and the fact that it has survived intact across the lineage. The disciplines do not change between phases; only the maturity of what they are working on changes.

A NASA-style Sample Retrieval Lander schedule (Figure 1) makes the distinction clear. The schedule is organized into four phases: Phase A (Concept and Technology Development), Phase B (Preliminary Design and Requirements), Phase C (Final Design and Fabrication), and Phase D (Assembly, Integration, Test, and Launch). Each phase is gated by a formal completion milestone, and the phases proceed in order, culminating in launch. Reading down a single column, the plan looks like Waterfall.



	Task Name	Low Dur	Base Dur	High Dur	Start	Finish	Resource	Predecessors
1	**SLR Project**		**235.4 wk**		**01/06/25 08:0**	**07/10/29 17:0**		
2	Program Manager		**235.4 wk**		**01/06/25 08:0**	**07/10/29 17:0**	**Program Office**	
3	**Phase A — Cc**		**50.2 wk**		**01/06/25 08:0**	**12/22/25 17:0**		
4	SRL Flight S	46 wk	46 wk	46 wk	01/06/25 08:0	11/21/25 17:0	Flight System IPT	
5	EDL Syster	41.8 wk	41.8 wk	41.8 wk	01/06/25 08:0	10/23/25 17:0	EDL IPT	
6	Sample Acq	33.4 wk	33.4 wk	33.4 wk	01/06/25 08:0	08/26/25 17:0	Sample Handling IP	
7	MAV Integra	33.4 wk	33.4 wk	33.4 wk	01/06/25 08:0	08/26/25 17:0	MAV Integration IPT	
8	Ground Sys	25 wk	25 wk	25 wk	01/06/25 08:0	06/27/25 17:0	MOS / GDS	
9	System Tes	50.2 wk	50.2 wk	50.2 wk	01/06/25 08:0	12/22/25 17:0	I&T / V&V Lead	
10	Mission & S	50.2 wk	50.2 wk	50.2 wk	01/06/25 08:0	12/22/25 17:0	Science Office	
11	Phase A Cc	0 wk	0 wk	0 wk	12/22/25 17:0	12/22/25 17:0		4,5,6,7,8,9,10
12	**Phase B — Pr**		**100.6 wk**		**12/23/25 08:0**	**11/25/27 17:0**		**11**
13	SRL Flight S	54.4 wk	100 wk	150 wk	12/23/25 08:0	11/22/27 17:0	Flight System IPT	
14	EDL Syster	54.4 wk	100 wk	150 wk	12/23/25 08:0	11/22/27 17:0	EDL IPT	
15	Sample Acq	50.2 wk	100 wk	150 wk	12/23/25 08:0	11/22/27 17:0	Sample Handling IP	
16	MAV Integra	50.2 wk	100 wk	150 wk	12/23/25 08:0	11/22/27 17:0	MAV Integration IPT	
17	Ground Sys	37.6 wk	100 wk	150.6 wk	12/23/25 08:0	11/22/27 17:0	MOS / GDS	
18	System Tes	58.6 wk	100.6 wk	150 wk	12/23/25 08:0	11/25/27 17:0	I&T / V&V Lead	
19	Mission & S	58.6 wk	100.6 wk	120 wk	12/23/25 08:0	11/25/27 17:0	Science Office	
20	Phase B Cc	0 wk	0 wk	0 wk	11/25/27 17:0	11/25/27 17:0		13,14,15,16,17,
21	**Phase C — Fi**		**37.6 wk**		**11/26/27 08:0**	**08/15/28 17:0**		**20**
22	SRL Flight S	37.6 wk	37.6 wk	37.6 wk	11/26/27 08:0	08/15/28 17:0	Flight System IPT	
23	EDL Syster	37.6 wk	37.6 wk	37.6 wk	11/26/27 08:0	08/15/28 17:0	EDL IPT	
24	Sample Acq	33.4 wk	33.4 wk	33.4 wk	11/26/27 08:0	07/17/28 17:0	Sample Handling IP	
25	MAV Integra	33.4 wk	33.4 wk	33.4 wk	11/26/27 08:0	07/17/28 17:0	MAV Integration IPT	
26	Ground Sys	29.2 wk	29.2 wk	29.2 wk	11/26/27 08:0	06/16/28 17:0	MOS / GDS	
27	System Tes	37.6 wk	37.6 wk	37.6 wk	11/26/27 08:0	08/15/28 17:0	I&T / V&V Lead	
28	Mission & S	33.4 wk	33.4 wk	33.4 wk	11/26/27 08:0	07/17/28 17:0	Science Office	
29	Phase C Cc	0 wk	0 wk	0 wk	08/15/28 17:0	08/15/28 17:0		22,23,24,25,26,
30	**Phase D — As**		**46 wk**		**08/16/28 08:0**	**07/03/29 17:0**		**29**
31	Project Mgr	46 wk	46 wk	46 wk	08/16/28 08:0	07/03/29 17:0	Program Office	
32	SRL Flight S	41.8 wk	41.8 wk	41.8 wk	08/16/28 08:0	06/04/29 17:0	Flight System IPT	
33	EDL Syster	41.8 wk	41.8 wk	41.8 wk	08/16/28 08:0	06/04/29 17:0	EDL IPT	
34	Sample Acq	37.6 wk	37.6 wk	37.6 wk	08/16/28 08:0	05/04/29 17:0	Sample Handling IP	
35	MAV Integra	37.6 wk	37.6 wk	37.6 wk	08/16/28 08:0	05/04/29 17:0	MAV Integration IPT	
36	Ground Sys	46 wk	46 wk	46 wk	08/16/28 08:0	07/03/29 17:0	MOS / GDS	
37	System Tes	46 wk	46 wk	46 wk	08/16/28 08:0	07/03/29 17:0	I&T / V&V Lead	
38	Phase D cc	0 wk	0 wk	0 wk	07/03/29 17:0	07/03/29 17:0		31,32,33,34,37
39	Launch	0.8 wk	1 wk	1.2 wk	07/04/29 08:0	07/10/29 17:0		38

Figure 1. Sample Retrieval Lander schedule (notional).

Read across the rows, it is not. The same seven Integrated Product Teams — Flight System; Entry, Descent, and Landing; Sample Acquisition and Transfer; MAV Integration; Mission Operations and Ground Data Systems; Integration and Test / Verification and Validation; and Science — appear in every phase. The IPTs are not handoff stations; they are durable, cross-functional teams that carry their capability through the entire program. The Flight System team in Phase A is the same as the Flight System team in Phase D; what changes is the maturity of the Flight System capability, not which discipline holds the work.

This structure directly addresses Waterfall's stovepiping. There is no disciplinary handoff to freeze: each IPT is cross-disciplinary and persistent. There is no phase gate at which a discipline ships its frozen output downstream and exits the program; the same team that wrote the Phase A concept for Entry, Descent, and Landing is responsible for its Phase D integration. Learning that surfaces in a later phase flow back to the IPT that owns the capability, because that IPT never left. The phase boundaries determine the level of understanding, not ownership of the work.

What makes the maturity gates substantive rather than ceremonial is the breadth of stakeholders engaged at each major phase transition. A NASA-style Key Decision Point is not a team milestone reviewed within the program; it is a formal commitment audited by every party with standing in the program's success. At each transition — Key Decision Point A, B, C, D, and E in NASA's nomenclature — the program team, the responsible Mission Directorate, the Office of the Chief Engineer, the Office of Safety and Mission Assurance, an independent Standing Review Board, the science community, mission operations, the launch services provider, applicable international and interagency partners, and the Agency Decision Authority all participate. Each stakeholder examines the capability set at its current maturity against the criteria for the next maturity state and holds authority to require changes, demand additional risk retirement, or withhold approval to proceed. The phase transition is a multi-stakeholder ratification that the program's understanding of every capability — and of the risks each carries — has reached the level required for the next phase of investment. This is how a phased plan avoids becoming a sequence of single-actor handoffs: every gate is owned by everyone with a legitimate stake in the outcome.

The schedule also illustrates the role of risk in CBP planning. Each task carries three duration estimates — Low, Base, and High — making uncertainty visible in the plan rather than buried behind a single deterministic date. The widest spread appears in Phase B, where the principal architectural and requirements risks are being retired (Flight System B and EDL B each span 54.4 to 150 weeks). By Phase D, the spreads are tight because the principal uncertainties have been resolved. The plan is structurally a statement about which capabilities carry which risks at which stage of maturity — which is what a CBP plan is supposed to be.

The schedule also makes visible what the lineage chapter argued in the abstract: the deterministic production-optimization methods misapplied to development have a legitimate home elsewhere in the program's life. The SRL plan ends at launch, but the program does not. Phase E — operations and sustainment — begins the moment the vehicle reaches its destination, and it is a materially different kind of work. The spacecraft exists. Operational procedures exist. The telemetry stream is continuous. Anomaly responses follow established protocols. Software and capability updates are discrete units of known scope. The work is no longer discovery; it is sustainment. In that regime, the production-oriented methods — throughput tracking, cycle time, defect rates per unit, queue management, and Reinertsen's product-flow economics — become legitimately applicable because the conditions they assume hold. CBP and the other uncertainty-oriented planning methods organize Phases A through D, where the dominant problem is discovery under uncertainty. Production-oriented flow methods take over for Phase E, where the dominant problem is the efficient flow of well-defined work against a deployed product. Different regimes, different methods, neither competing with the other.

More broadly, the SRL example shows that CBP is not a rejection of staged planning. Staged planning, in the maturity-of-understanding sense RUP introduced, is among the lineage's most durable survivors. What CBP does not retain from Waterfall is the stovepiped, discipline-frozen, handoff-driven aspect of phased work; what it adds is the use of capability, rather than discipline, as the unit around which the work is organized.

Practitioners working with CBP describe the contrast with Waterfall in a single image. Waterfall, as noted earlier, is the project-management equivalent of a ballistic missile: once launched, its trajectory is

fixed and it cannot track a moving target. CBP, by their account, is more like a cruise missile. It has a target, a flight plan, and the means to make corrections in flight. Each phase boundary serves as a guidance update — a multi-stakeholder review in which the current state of every capability is checked against the target, named assumptions are confirmed or invalidated against evidence accumulated since the last review, and the plan is adjusted accordingly. Between phase boundaries, the IPTs continuously monitor their capabilities, raising issues against named assumptions as soon as the evidence warrants. The risk analysis structure described in the next section functions as the guidance system, indicating where the program is relative to the target, which deviations matter most, and which corrections close the gap most efficiently. The result, in the practitioner's framing, is a plan that can steer.

4.5 Digital Twins as the Substrate for Learning

If the central principle of project management for novel work is the systematic reduction of uncertainty through learning, the operational question is how to make learning fast, frequent, and cheap enough to keep pace with the program. Physical-world learning is slow and expensive: a flight qualification test costs tens of millions of dollars and takes months to set up; a launch failure is informative but catastrophically so; a wind tunnel campaign retires one assumption at a time. Programs using the lineage's earlier methodologies were forced to plan around this scarcity, reserving learning for a few discrete milestones — a critical design review, an integration test, a first flight — and accepting that priors between those events would drift further and further from the underlying reality.

Digital twins materially change that economics. The National Academies define a digital twin as a set of virtual information constructs that mimic the structure, context, and behavior of a natural, engineered, or social system, are dynamically updated with data from its physical counterpart, and have predictive capability that informs decisions and realizes value. The defining property is bidirectional synchronization: the twin is not a one-time model built at design time and abandoned, but a live representation that ingests telemetry, test data, manufacturing variation, and operational evidence throughout the program and continues to predict what the physical artifact will do under conditions it has not yet experienced. In CBP terms, a digital twin is a learning machine operating on the same units as the plan — capabilities, performance against required thresholds, and the assumptions on which each depends.

Institutional adoption has moved quickly. The Department of Defense codified digital engineering in its 2018 Digital Engineering Strategy, updated in December 2023, and DoD Instruction 5000.97 (2024) requires new acquisition programs to specify their digital engineering ecosystem, digital models (including digital twins), digital threads, and digital artifacts as part of the acquisition strategy and systems engineering plan. NASA has run digital twin pilots on Orion and the Space Launch System, and the ARRISTOTLE high-fidelity simulator runs the actual flight software for SLS and Orion against simulated Kennedy Space Center ground systems, verifying mission scenarios and crew safety in advance of the physical campaign. Industry adoption in commercial aerospace, automotive, and, increasingly, in pharmaceutical and infrastructure development follows similar patterns: a twin is built early, refined continuously, and treated as a working model of the physical system that can be queried at the cost of compute rather than hardware.

The connection to the principle is direct. Every assumption in the capability model is a hypothesis, and every hypothesis is, in principle, testable. A digital twin makes a large fraction of those hypotheses testable in software, in hours rather than months, at compute cost rather than qualification cost. An aerodynamic margin can be exercised against a sweep of off-nominal conditions before a single physical surface is fabricated. A propulsion system's thermal margin can be checked against thousands of trajectory perturbations before a ground test. A software control law can be verified against the actual flight software in a simulated environment that includes the ground system, the operators, and failure

modes the physical campaign cannot economically reproduce. Each exercise sharpens Bayesian priors on the affected capability — narrows the performance distribution, confirms or invalidates a named assumption, and retires or activates a risk event on the register. The twin is the engine that makes learning continuous rather than episodic.

The Mars Climate Orbiter failure described earlier illustrates what a digital twin detects that V-Model verification does not. The MCO defect lay in the interface specification between two subsystems: one delivered force in pound-force-seconds, while the other expected newton-seconds. Each subsystem verified faithfully against its agreed inputs. V-Model verification at every level passed because each level checked the implementation against the baseline that contained the defect. A digital twin that ran the actual subsystem outputs — the real propulsion command, the real navigation update — through a flight-dynamics model of the spacecraft and the planet would have computed a trajectory different from the design intent, because the physics does not care which units the spec is written in; it cares about the numbers the subsystems actually exchange. The discrepancy would have been visible long before launch, in routine Bayesian updating of the navigation capability's posterior. The MCO loss is, in this sense, a case of an unbuilt digital twin: the failure was preventable by an integration mechanism that operates on physics rather than on specifications. A V-Model can verify that a system implements its spec; a digital twin can verify that the system, as specified and implemented, will do what the mission requires. The two are complementary, and the MCO record is what it looks like when the second is missing.

Digital twins also support the multi-stakeholder review structure that makes CBP's phase gates substantive. At a key decision point, the program team, the engineering review boards, the safety and mission assurance organizations, the independent review board, the science community, mission operations, the launch services provider, and the decision authority can interrogate the same twin and see the same predicted behavior of the capability set. Disagreements about residual risk become disagreements about specific model assumptions and evidence streams, which can be examined and adjudicated. A capability marked at sixty percent of its required performance is sixty percent of the way to its required performance against a model that every stakeholder can probe, not against an assertion that one organization makes, and another organization is asked to accept.

Three cautions follow from current experience. First, a digital twin is only as good as the model and data that compose it; an unvalidated twin can generate confidence faster than the evidence justifies. Verifying twin behavior against the physical system is a discipline in itself, and recent academic work has begun to apply formal methods — including the Temporal Logic of Actions — to specify and verify twin correctness. Second, the twin's value scales with the program's discipline in keeping it current. A twin that lags the physical artifact, ignores manufacturing variation, or operates on stale parameter sets quickly becomes a source of false comfort. Third, twins are not a substitute for physical testing in regimes where the physics is poorly modeled or the consequences of being wrong are severe; they shift the balance between physical and virtual evidence but do not eliminate the requirement for the former.

Taken together, digital twins are an operational technology that makes CBP's principle of continuous learning practical at program scale. They are not part of the methodology as originally articulated — capability-based planning predates the current digital-engineering infrastructure by more than a decade — but they are increasingly the substrate on which current institutional practice runs. The CBP–digital twin pairing is one approach to the shortfalls the lineage exposed, and the one most clearly codified at institutional scale today; other approaches exist, and the field's experience with this pairing is still accumulating. What is established is that a CBP plan running over a mature digital twin is the closest the lineage has come to a system in which learning is structurally part of the plan rather than a value the plan invokes.

5. How CBP Supports Risk Management and Risk Analysis

Risk management and risk analysis are distinct disciplines, often conflated in casual use. Risk management identifies risk events — specific occurrences that may happen during a program, each characterized by a probability of occurrence and an impact on the goal — and tracks, mitigates, and retires them over time. Risk analysis is the quantitative discipline that propagates the plan's underlying uncertainties, conditional on the identified risk events, into a posterior distribution over whether the program will meet its objective. Risk management produces a managed inventory of named risks. Risk analysis produces a probability, with a confidence interval, that the program will succeed. The two are tightly coupled — analysis is conditional on the events management has identified — but they are not the same activity. Prior methodologies in the lineage tended to support one or the other, rarely both, and rarely well.

CBP's plan structure provides the substrate for both. The capability model identifies the units of value, the assumptions each depends on, and the dependencies between units. That structure makes risk events well-defined and probabilistic analysis tractable. The two subsections that follow describe how each discipline operates against a CBP plan.

5.1 Risk Management Against the Capability Model

Under CBP, risk management operates against the capability model rather than against a task breakdown. A risk event is a named contingency tied to one or more capabilities: a key assumption may fail to hold, a required technology may not mature on schedule, a dependency may not be delivered, a regulator may decline approval, a test article may fail. Each event is characterized by two random variables — probability of occurrence, and impact on the goal — both expressed as distributions rather than point estimates, because both are themselves uncertain. The risk register is a structured collection of these named events, each tied to the capability or capabilities it threatens.

This structure follows directly from the capability model and is not available without it. Anchoring risk events to capabilities answers the questions risk management has long struggled with: what does this risk threaten, and what would have to be true for the program to succeed despite it? In a backlog-driven or task-driven plan, a risk event is typically associated with a piece of work, and the connection between that work and the program's goal is left informal. In a capability plan, a risk event is associated with a capability whose contribution to the outcome is already in the model. The risk register is, in effect, an overlay on the capability map: where are the named events, which capabilities do they threaten, and which outcomes are exposed through those capabilities?

Risk management within a capability model has a natural lifecycle. Risk events are born when identified, mature as evidence accumulates on their probability and impact, are mitigated when actions reduce either, and are retired when the contingency they describe is resolved — favorably, when the assumption holds and the threat passes, or unfavorably, when the event occurs and the consequences become known. Each phase gate is, among other things, a review of the risk register: which events have been retired, which have changed in probability or impact, and which new events have emerged.

5.2 Risk Analysis: Bayesian Updating Against the Objective Function

Risk analysis under CBP takes the plan's underlying uncertain quantities and, conditional on the risk events identified by the management discipline, propagates them into a posterior distribution over the objective function. The underlying quantities are the priors: three-point estimates for each task or work package, distributions for each cost element, distributions for the performance each capability will ultimately achieve relative to its required threshold, and prior probabilities for the named assumptions supporting each capability. The objective function is the program's measure of success — probability of

mission success, projected return on investment, probability of closing a capability gap by a target date, or operational readiness by a deployment window.

The relationship between the risk register and the analysis is conditional. Each named risk event updates the priors for the affected capabilities: if the event occurs, the duration distribution shifts right, the cost distribution shifts up, the performance distribution shifts down, or some combination. The posterior on the objective function is therefore a probability-weighted mixture over the joint distribution of risk events. Monte Carlo simulation is the standard tool for computing this mixture because closed-form integration becomes intractable once the number of dependent capabilities and risk events exceeds a handful. The simulation samples from the joint distribution of priors and risk events, propagates each sample through the capability dependencies, and produces an empirical posterior on the objective function.

Bayesian updating keeps the analysis current. As evidence accumulates — a test result, a successful integration, a missed milestone, a confirmed supplier delivery, a regulator's preliminary finding — the priors are revised. A successful qualification test sharpens the performance distribution for the tested capability. A missed milestone widens the duration distribution for the affected work and may increase the probability of a downstream risk event. A confirmed assumption removes a risk event from the register and narrows the posterior. Where a program maintains a digital twin (Section 4.5), the twin is a continuous source of such evidence: simulated runs under off-nominal conditions, parameter sweeps across the design space and replay of operational telemetry all feed updates to the priors at a frequency far beyond what physical testing alone supports. The objective function is recomputed using the updated priors, and the program's expected probability of success — with its confidence interval — shifts accordingly. This is not a one-time analysis but a continuous one, refreshed at every major phase review and whenever significant new evidence arrives.

The result is a plan whose top-line answer to "will this program succeed?" is a probability with a confidence interval, derived from priors on cost, duration, capability performance, and assumption validity, conditional on the risk register. That probability shifts as capabilities mature, assumptions are confirmed or falsified, risk events occur or retire, and costs and durations resolve. Executives, boards, and decision authorities see not only the current expected outcome but also its trajectory over time, and can identify the specific capabilities, assumptions, risk events, or cost elements driving any adverse trend.

Several techniques are commonly used within this framework. Joint Cost and Schedule Confidence Level (JCL) analysis, standard in NASA program management, produces the joint posterior for cost and schedule given the priors and the risk register. Sensitivity analysis identifies which capabilities, assumptions, or risk events contribute most to the variance in the objective function, directing risk-retirement attention where it has the most leverage. Decision-tree and real-options analysis evaluate phased commitments under uncertainty — the formal counterpart to the maturity-gate structure described earlier. None of these techniques is original to CBP. What is distinctive is that CBP's plan structure provides the inputs they require: a capability with required performance, current performance, cost, duration, and named assumptions, tied to a risk register of named events with probability and impact distributions, is already a model in the form the techniques expect. Prior methods did not present such a model, which is why risk analysis under them was typically a separately maintained activity attached to a plan that did not natively support it.

5.3 Objective Measurement of Progress Toward Success

A structural feature of CBP that follows directly from its capability-and-outcome model — and that did not appear in earlier methods — is that progress can be measured against an objective definition of project success. Each prior method measured activity, with later methods adding an outcome layer

above the activity layer. Waterfall measured phase completion. RUP measured artifact production. XP measured story completion and test pass rates. Agile and its scaling frameworks measure story points, velocity, sprint goals, features shipped, and program increment objectives. The production-oriented tradition measured throughput, cycle time, and defect rates. Each of these is a measure of work performed. The Agile community has since added outcome-oriented frameworks — OKRs, North Star Metrics, Evidence-Based Management, business agility instrumentation — that go beyond activity to measure the outcomes the team is committed to delivering. These outcome metrics are a real improvement over activity metrics, but they are still readings on the system's current state: they answer, "did the last cycle work?" rather than "will the program meet its objective?"

Under CBP, the situation is structurally different. The capability set is built backward from the outcome: the program's definition of success — mission success for a science or defense program, return on investment for a commercial product, regulatory approval for a compliance program, operational readiness for a fielded capability — is decomposed into the capabilities required to achieve it, each with stated performance thresholds and the assumptions on which it depends. By construction, progress against a capability is progress against the outcome. A capability measure answers a different question from an outcome metric: not "is the system in the state we wanted last quarter?" but "is the system on a trajectory to be in the state the enterprise needs at the program's target date?" The risk management and risk analysis disciplines described above operate on the same units. There is no translation step between "the team did this work," "this risk event is on the register," and "the project is closer to succeeding," because the unit of planning, the unit of risk, and the unit of value are the same.

Earlier frameworks did not provide this. A velocity-driven Agile program can deliver 8 sprints of strong results while the business sees no change in its position. A SAFe program can produce a steady cadence of program increments while the underlying outcome metrics remain flat. An OKR-driven program can hit its key results for several quarters while the capability gap the enterprise actually needs to close keeps widening, because OKRs measure where the system is, not what it must become. Such frameworks lack a mechanism to detect the disconnect because their measurements reach the activity or outcome layer but not the capability layer, and their risk registers, when they exist, are anchored to the work rather than the goal. CBP closes both gaps by construction: a capability that is 60% of the way to its required performance level corresponds to 60% progress toward the outcome it underwrites, and a risk event tied to that capability is a risk to that outcome.

A CBP plan can therefore answer, at any point in its execution, four questions that prior methods could only address by proxy: which capabilities are on track to deliver the outcomes the program is committed to; which are at risk and by how much; which named risk events are open and what their current probability and impact are; and what the current posterior probability of mission success or projected return is, given the present state of capability maturity and the open risk register. These are the questions executives, boards, and customers commonly ask. CBP is the first methodology in the lineage in which those questions have answers that derive from the plan itself rather than from a separately maintained dashboard attached to a plan that does not speak the same language.

5.4 Comparative Summary

Methodology	Primary Risk Managed	Risk Left Unmanaged	Level of Operation
Waterfall / IBM Phased	Schedule, scope, contract	Change, late discovery	Project

Methodology	Primary Risk Managed	Risk Left Unmanaged	Level of Operation
V-Model (INCOSE)	Verification against baseline; SE discipline	Verifies wrong baseline; no course correction	Project / system
Spiral	Iteration risk	No definition of done; runaway iteration	Project
RUP	Architectural risk; staged maturity	Prescriptive artifact load; ceremony	Project
XP	Inner-loop engineering quality	Business planning; choosing what to build	Team
Software Factory	Process repeatability, defect rates	Discovery; category error vs. manufacturing	Organization
Lean	Waste, flow inefficiency	Discovery; inherits the factory's category error	Value stream
Agile	Team-level delivery and adaptation	Business planning; backlog validity	Team / program
Capability-Based Planning	Plan risk, outcome risk, assumption risk	Requires complementary delivery method	Program / plan

6. Conclusion

The history of project management is a history of methodologies, each with a fundamental limitation in its governing assumptions, each exposed when extended beyond its proper regime, each corrected by the next, and each leaving behind a residue of practices that remain in use. Test-driven development, continuous integration, refactoring, short iterations, daily stand-ups, retrospectives, staged maturity of understanding, and product-flow economics for change-request streams are all survivors of this process. They appear in current practice wherever the conditions for which they were designed hold.

What did not survive was the universalizing claim each methodology made about itself. None proved to be an answer to the question that persisted across the lineage: how is the validity of the plan itself to be analyzed? In many consequential failures, the team executed the plan faithfully; what failed was the plan.

Capability-Based Planning, particularly when paired with the digital engineering and digital twin infrastructure now codified in institutional practice, is one response — not the only viable response, but the one most clearly visible in current institutional adoption. It is the planning and control layer that

earlier methods lacked, and the historical record shows it operating alongside the surviving practices below it rather than in place of them. CBP treats the plan itself as an object of analysis. It expresses required outcomes, the capabilities that produce them, and the assumptions on which they depend, in a form that can be audited, challenged, and corrected. Because the capability set is built backward from the outcome, it measures progress against an objective definition of project success — mission success, return on investment, operational readiness, regulatory approval — rather than against proxies for activity. The same structure supports both risk management — the identification and tracking of named risk events, with their probability and impact distributions, against the capabilities they threaten — and risk analysis — the quantitative propagation of priors on cost, duration, and capability performance, conditional on the risk register, into a posterior probability of meeting the objective, updated by Bayesian methods as evidence accumulates, with the digital twin serving as a continuous source of that evidence. CBP is, in the lineage this paper traces, the first methodology in which these properties appear together in the plan itself.

CBP, taken with the production-oriented methods that govern operations and sustainment after delivery, forms the current integration of the lineage's lessons. CBP and other uncertainty-oriented planning methods organize work where the dominant problem is discovery under uncertainty — concept, design, fabrication, integration, and delivery. Production-oriented flow methods take over after delivery, when the work is repetitive, the artifact is defined, and throughput and cycle time apply. The surviving engineering and team practices — test-driven development, continuous integration, refactoring, short iterations, daily stand-ups, retrospectives — operate within both regimes. None proved universal, and the record suggests caution against treating any as universal in the future. Each is in the place its experience has earned. That is the lesson the lineage has taken seventy years to teach, and the lesson on which current institutional practice — at NASA, at the U.S. Department of Defense, and at organizations adopting their methods — is now being built.

Although the lineage developed largely within software, the pattern is not limited to it. The same dynamic operates wherever novel systems are developed under uncertainty: defense acquisition, spacecraft and aerospace programs, pharmaceutical research, infrastructure megaprojects, and large-scale organizational transformation. In each case, the dominant failure mode is not the team's inability to execute the plan, but the plan, faithfully executed, producing an outcome the enterprise no longer needed or never needed. The retained practices — staged maturity of understanding, durable cross-functional teams, lightweight coordination ceremonies, probabilistic risk analysis — and the layer above them, where plan validity is analyzed, apply to any novel system development that takes its uncertainty seriously. Software is one instance of that broader category, not the category itself.

The reasoning chain with which this paper opened — novelty implies incomplete information, incomplete information is uncertainty, and uncertainty is risk — directly implies the principle the lineage took seventy years to discover and which CBP makes structural: the central activity of project management for novel work is learning and uncertainty reduction, and the plan exists to organize and accelerate that activity rather than to substitute for it. A methodology earns its place to the extent that it honors this principle. Every surviving practice in the lineage — short iterations, continuous integration, retrospectives, maturity gates, three-point estimates, Bayesian updating, capability-based planning — honors it in a specific way. Every failed methodology violated it in a specific way. The history is, in the end, the history of a field learning to take its own uncertainty seriously.