International Journal of Computer Techniques–IJCT Volume 9 Issue 2, May - 2022

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Agile Methodologies for Data Engineering Teams: Adoption Patterns and Outcomes

A Survey with Practitioner Case Studies

Kuladeep Sandra

Independent Researcher

kuladeepsandra90@gmail.com

Abstract*—Agile methods, articulated in the Agile Manifesto and operationalized in frameworks including Scrum, Extreme Programming, and Kanban, have been the dominant approach to organizing software engineering work for nearly two decades. Their adoption in data engineering has been slower and more uneven, and the explanation is not principally cultural. Software engineering Agile rests on conditions (clear acceptance criteria, testable outputs, deployment independence, low cross-team coupling) that data engineering only partially satisfies. Data engineering work involves infrastructure dependencies that block teams unpredictably, exploratory analyses whose duration cannot be bounded in advance, shared platform components whose changes affect all consumers simultaneously, and quality failures that surface weeks after the sprint in which the responsible code shipped. Teams that import software Agile patterns unchanged into data engineering settings tend to fail in characteristic ways: estimates shatter against unknowable infrastructure complexity, sprints become ceremonies that document the inability to plan rather than the ability to deliver, and morale erodes as the team learns to discount its own commitments. This paper surveys the Agile adoption patterns that have emerged in data engineering, the data-specific challenges that drive adaptation, and the measurable outcomes that adapted patterns produce. It addresses three lines of inquiry: (RQ1) what Agile adoption patterns exist in data engineering teams and how do these patterns emerge from the unique characteristics of data engineering work; (RQ2) what challenges are specific to Agile adoption in data engineering and what adaptations prove effective; and (RQ3) what measurable outcomes result from Agile adoption and how do teams reconcile productivity gains with the persistence of estimation challenges. The paper combines a structured survey of literature published through 2022 with two longitudinal practitioner case studies: a 2018 sprint failure (Sprint 7) in which over-commitment and inadequate risk identification produced a goal completion rate of around 42 percent and a painful retrospective; and a 2019–2020 platform enablement squad initiative that achieved about a 20 percent efficiency gain (measured as the ratio of story points delivered per sprint-week before and after the reorganization, averaged over six sprints) in domain team velocity at cost-neutral first-year economics. The closing argument is that Agile works for data engineering when its patterns are adapted intentionally to the conditions of data work; teams that adopt software Agile unchanged should expect failures of the kind documented here, and the failures are recoverable through the kinds of adaptations the case studies illustrate.

***Index Terms*—Agile, Scrum, Kanban, data engineering, team organization, productivity, Agile adoption**

I. INTRODUCTION

describes. The model combines Scrum at the domain team level (where feature delivery is the dominant work) with Kanban at the platform team level (where infrastructure work flows continuously and does not respect sprint boundaries), with explicit service level agreements between the two, with a platform enablement squad pattern that embeds platform engineers with domain teams, with quarterly outcome commitments rather than sprint-level forecasts, and with several smaller adaptations including range estimation, risk backlogs, and mid-sprint replanning. The model is not novel in its individual elements; what is reported here is the integration of those elements in a data engineering context and the outcomes that the integration produced.

This paper surveys the Agile adoption patterns that have emerged in data

A. Background

characteristics of the domain that made the methodology workable. Software features can usually be specified with clear acceptance criteria; automated tests provide binary feedback on whether a feature is done; deployment automation allows incremental rollout of features without coordinated release events; and a team of eight to twelve engineers can typically own a feature from design through production with limited dependencies on other teams. None of these characteristics is universal in software engineering, but they are common enough that the standard Agile patterns work in many software contexts.

Data engineering is a different kind of work, and its differences are consequential for Agile adoption. Scope is often ambiguous in ways that software scope is not: a request to build a customer 360 view requires discovery of which fields are needed, how many sources are involved, and what quality bars are acceptable, and the answers emerge through exploration rather than at sprint planning. Infrastructure dependencies are pervasive: domain teams depend on platform teams for Kubernetes capacity, Kafka

topics, storage, schema registry updates, and governance policies, and platform team delays cascade into domain team sprint failures. Shared infrastructure means that platform changes (a Spark version upgrade, a schema registry change) affect all consuming teams simultaneously, requiring coordination overhead that single-team Agile assumptions do not anticipate. Quality failures surface long after deployment: a pipeline that completes successfully on day 5 of a sprint may produce subtly wrong data that is only noticed weeks later when an analyst spots a discrepancy. Exploratory work coexists with delivery work in ways that resist sprint boundaries: an exploratory data analysis might finish in three days or might run for a month, and trying to commit to either outcome during sprint planning produces either dishonest planning or wasted capacity.

The author's experience with Agile in data engineering contexts began in 2012 and proceeded through several waves of adoption, adaptation, and failure. The earliest attempts imported software Scrum unchanged. Story points for infrastructure work proved wildly unreliable. Sprint plans were fantasy documents whose contents bore limited resemblance to what the team actually worked on. The decisive failure came in what this paper calls Sprint 7 (the events are described in detail in Section 7), in which the team committed to a sprint goal that combined a Kafka cluster migration with new data source integration, encountered cascading infrastructure issues that the planning had not anticipated, and missed the sprint goal by a margin large enough to demoralize the team and to force an honest reckoning with the limits of naive Agile in data work.

The recovery from that failure produced the hybrid model that this paper

B. Research Questions

engineering teams, the challenges that drive adaptation, and the outcomes that adapted patterns can produce. The survey is anchored by three lines of inquiry:

RQ1: What Agile adoption patterns exist in data engineering teams, and how do these patterns emerge from the unique characteristics of data engineering work, including infrastructure dependencies, exploratory uncertainty, and tight cross-team couplings?

RQ2: What challenges are specific to Agile adoption in data engineering, as distinct from software engineering, and what adaptations prove effective in addressing them?

RQ3: What measurable outcomes result from Agile adoption in data engineering teams, and how do teams reconcile productivity gains with the persistence of estimation challenges and project surprises?

C. Paper Organization

engineering teams and who are navigating their own Agile adoption decisions, and researchers in software engineering and information systems who study team productivity and methodology adoption in non-traditional contexts. The paper is not a recommendation of one Agile framework over another and is not a tutorial on Scrum or Kanban; it assumes basic familiarity with the principal frameworks and focuses on the choices and adaptations that data engineering work specifically requires.

The remainder of the paper is organized as follows. Section 2 reviews the foundations of Agile in software engineering and identifies the characteristics of data engineering that create friction with software-derived patterns. Section 3 describes the survey methodology. Section 4 surveys the principal Agile adoption patterns observed in data engineering teams. Section 5 examines the challenges unique to data engineering and the adaptations that have proved effective. Section 6 presents the Sprint 7 failure case study in detail, including the events, the retrospective findings, and the recovery. Section 7 presents the platform enablement squad case study, including the pilot, the rollout, and the quantified outcomes. Section 8 discusses open challenges and future research directions. Section 9 concludes.

II. BACKGROUND: AGILE IN SOFTWARE VS DATA ENGINEERING

==========

This section reviews the foundations of Agile in software engineering and identifies the characteristics of data engineering that create friction with software-derived adoption. The objective is to ground the pattern survey in Section 4 and the challenge analysis in Section 5 in the underlying reasons that data engineering adoption follows different paths.

2.1 Agile Foundations

A. Agile Foundations

The Agile Manifesto (Beck et al., 2001) proposed four value statements: 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. The manifesto was supplemented by twelve principles emphasizing iterative delivery, frequent customer feedback, self-organizing teams, sustainable pace, and embracing change. Frameworks operationalized these principles in different ways. Scrum (Schwaber and Beedle, 2002) defined a process structure of fixed-length sprints, defined roles (product owner, scrum master, development team), and ceremonies (sprint planning, daily standup, sprint review, retrospective). Extreme Programming (Beck, 2000) emphasized engineering practices including test-driven development, pair programming, continuous integration, and refactoring. Kanban (Anderson, 2010) emphasized visualization of work, work-in-progress (WIP) limits, and continuous flow rather than time-boxed iterations. Lean software development (Poppendieck and Poppendieck, 2003) framed software development in terms of value streams and waste elimination borrowed from manufacturing.

Subsequent work refined the frameworks for larger organizations. The Scaled Agile Framework, Disciplined Agile Delivery, Large-Scale Scrum, and the Spotify model offered different approaches to coordinating multiple teams under Agile principles. Skelton and Pais (2019) proposed the Team Topologies framework that identifies four fundamental team types (stream-aligned, enabling, complicated subsystem, and platform) and three interaction modes between them, providing a vocabulary that is particularly useful for the platform enablement squad pattern discussed in Section 7.

B. Conditions That Make Software Agile Workable

Several characteristics of typical software engineering work make standard Agile patterns workable. First, scope is often clear at the level of individual features. A request to implement a login feature has acceptance criteria that the product owner and the team can agree on at sprint planning: the user can authenticate, the session persists, logout works, password reset works. Second, testability is binary: automated tests verify that the feature meets its specification, and the tests pass or fail without ambiguity. Third, deployment independence is generally achievable: features can be released behind toggles, gradually rolled out, and rolled back if necessary, without coordinating with other features being developed in parallel. Fourth, team autonomy is feasible: a cross-functional team of eight to twelve engineers can typically own a feature from design through production with limited dependencies on other teams that would block its progress.

None of these characteristics is universal in software engineering. Embedded systems, safety-critical software, and tightly coupled enterprise systems all violate one or more of them, and Agile adoption in those contexts has historically been more difficult than in web application development. The point is that the conditions are common enough in mainstream software engineering to make standard Agile patterns the default workable approach.

C. Conditions That Create Friction in Data Engineering

Data engineering work routinely violates the conditions that make software Agile workable. Six sources of friction recur.

Scope ambiguity. A request to build a customer 360 view, or a churn prediction pipeline, or a regulatory reporting dataset, cannot be fully specified at sprint planning. The team must discover which source systems contain the relevant data, what state those systems are in, what definitional conflicts exist between business units, what quality bars the consumer requires, and what infrastructure capabilities the platform supports. The discovery itself is work, and the work cannot be planned without doing it.

Infrastructure dependencies. Domain team work depends on platform-team-owned components: Kubernetes capacity, Kafka topics and clusters, storage volumes, schema registry entries, governance policies. When the platform team has not delivered a required component, the domain team is blocked, and the blockage cannot be resolved through individual heroics within the domain team. Cross-team dependencies are pervasive in a way that cross-team dependencies in software engineering, where teams are usually decoupled architecturally, are not.

Shared infrastructure. A Spark version upgrade affects every team using Spark. A schema registry change affects every consumer of every schema. A storage layout change affects every workload that reads or writes to the affected paths. The blast radius of platform changes is wide, and the coordination overhead of managing that blast radius is real. Single-team Agile assumes that a team's choices affect only that team; in data engineering this assumption fails routinely.

Long-tailed execution. A data pipeline that completes successfully and loads data into a target table is not necessarily a working pipeline. The data may be wrong in subtle ways that surface only when an analyst notices a discrepancy, or when a downstream consumer derives an incorrect aggregate, or when an audit reveals a compliance issue. The interval between deployment and the surfacing of failure is often longer than the sprint that produced the deployment, which means that the sprint review cannot reliably evaluate whether a data product is actually working.

Exploratory versus delivery work. Data engineering includes both bounded delivery work (build the daily refresh of the revenue dashboard) and unbounded exploratory work (figure out which features predict customer lifetime value). Exploratory work resists time-boxing in a way that delivery work does not: a fruitful exploration might finish in a week, or might require a month, or might produce a negative result that is itself valuable. Mixing exploratory and delivery work in the same sprint produces consistent friction, with one or the other suffering when the other absorbs more time than planned.

Cross-team blocking dependencies. A typical analytics deliverable depends on the data engineering team's curated tables, which depend on the platform team's lakehouse infrastructure, which depends on the source system team's API or change-data-capture feed. Each of these teams operates on its own cadence, and the blocking dependencies between them create queues that no single team can plan its way out of. Traditional project management tools (critical path analysis, program evaluation and review technique) were designed for environments with this kind of dependency density; Agile frameworks generally assume that teams are decoupled enough to plan independently.

D. What Working Data Looks Like

Adapting Agile to data engineering requires translating the principle of \"working software\" into something appropriate to the domain. Working data is not simply data that has been delivered to a target table; it is data that meets quality SLAs (freshness, accuracy, completeness) at the moment of consumption and that continues to meet them through subsequent operations. The definition of done for a

data product must therefore include quality verification, and the verification must be automated to be sustainable. The case study in Section 7 illustrates how this principle was operationalized through Great Expectations integration in pipeline deployment.

Adapting customer feedback similarly requires recognizing that data customers are typically internal: analysts, machine learning engineers, business intelligence consumers, executives. The feedback loops can be fast (a daily dashboard whose users notice anomalies immediately) or slow (a quarterly business decision whose dependence on a particular dataset only becomes visible when the decision goes wrong). The team's process must accommodate both.

Self-organizing teams in data engineering require a different mix of skills than self-organizing software teams. A cross-functional data team may need a data engineer, an analyst, a domain expert, and possibly a machine learning engineer, and the relationships among these roles must be configured for collaboration rather than handoff. The case study site found that mixed-role teams produced better outcomes than role-segregated teams, though the integration required deliberate cultivation.

3. Methodology

III. METHODOLOGY

==========

This survey was conducted following a structured literature review approach (Kitchenham, 2004; Webster and Watson, 2002), supplemented by reflective analysis of practitioner case studies in which the author was a participant. The objective was to identify the Agile adoption patterns, challenges, and adaptations that characterize data engineering work and to validate the literature findings against documented production experience.

3.1 Scope and Sources

A. Scope and Sources

The literature search covered peer-reviewed venues including IEEE Software, the Journal of Systems and Software, the Empirical Software Engineering journal, the ACM Transactions on Software Engineering and Methodology, the proceedings of the International Conference on Software Engineering (ICSE), the proceedings of the Agile Conference, and the Information and Software Technology journal. Industry analyst publications (Gartner, Forrester) and practitioner books (Beck, Schwaber, Cohn, Anderson, Skelton and Pais) were included where they have shaped practice in ways that the peer-reviewed literature has not yet documented. All cited material is dated 2020 or earlier.

B. Search Strategy

Search strings combined the following terms in pairs and triples: (\"Agile\" OR \"Scrum\" OR \"Kanban\") AND (\"data engineering\" OR \"data pipeline\" OR \"analytics\"); \"Agile\" AND (\"infrastructure\" OR \"platform\") AND (team OR estimation OR adoption); (\"story points\" OR \"sprint planning\" OR \"backlog grooming\") AND (\"case study\" OR empirical); \"data mesh\" AND (Agile OR team OR organization); (\"team productivity\" OR velocity OR throughput) AND (Agile OR Scrum). Forward and backward citation tracing was applied to seminal works including Beck (2000), Schwaber and Beedle (2002), Cohn (2005), and Anderson (2010).

C. Inclusion and Exclusion Criteria

Publications were included if they (a) reported empirical case studies of Agile adoption in software, data, or platform engineering contexts; (b) proposed or evaluated patterns for team organization, estimation, or planning in iterative delivery contexts; or (c) addressed challenges of Agile adoption in non-traditional or large-scale environments. Publications were excluded if they were marketing collateral without empirical content, were narrowly focused on individual tool usage without broader methodological insight, or were opinion pieces without reported experience.

D. Classification Framework

Identified works were classified along three axes: the adoption pattern they addressed (pure Scrum, hybrid Scrum-Kanban, narrative sprints, strangler fig migration, data mesh with platform enablement); the challenges they engaged with (estimation accuracy, cross-team dependencies, exploratory versus delivery work, late-appearing failures, forecasting); and the outcomes they reported (velocity, sprint goal completion, cycle time, team satisfaction). The same axes structure the pattern survey in Section 4 and the challenge analysis in Section 5.

E. Practitioner Case Validation

To complement the literature analysis, the findings are mapped against two longitudinal case studies from the author's practice. The first case study is the Sprint 7 failure of 2018, in which a team of eight engineers committed to a sprint that combined infrastructure migration with new feature delivery and missed its goal by a substantial margin. The second case study is the platform enablement squad initiative of 2019--2020, in which embedded platform engineers were deployed across domain teams to reduce infrastructure blocking. Both case studies are presented in Sections 6 and 7 respectively, and both are presented as participant-observer reflection rather than as detached evaluation.

IV. SURVEY OF AGILE ADOPTION PATTERNS IN DATA ENGINEERING

==========

This section addresses RQ1 by surveying the principal Agile adoption patterns observed in data engineering teams. Five patterns are described: pure Scrum (the software-derived baseline), Scrum and Kanban hybrids (with

infrastructure work separated from feature work), narrative sprints (focused on outcomes rather than story-point output), strangler fig migration (for large-scale system replacement), and data mesh with platform enablement (the configuration that the case study site converged on). Table 1 summarizes the patterns and the conditions under which each works or fails; the discussion that follows treats each in turn.

Table 1. Agile adoption patterns observed in data engineering teams.

Pattern **Description** **Works When** **Fails When** ---------- ---------- ---------- ---------- Pure Scrum 2-week sprints, story-point estimation, all work in single backlog Small teams (3--5), stable infrastructure, narrow scope Cross-team dependencies; mixed exploratory and delivery work; infrastructure churn Scrum + Kanban hybrid Domain teams Scrum; platform team Kanban with SLAs Decoupling predictable feature work from unpredictable infrastructure Without explicit SLAs; when platform team becomes a bottleneck Narrative sprints Outcome-framed sprint goals; flexible scope inside the goal Cross-functional product teams; outcomes valued over output Reporting environments that demand point-level forecasting Strangler fig migration Old and new systems run in parallel; incremental traffic redirect Large migrations where flag-day cutover is unacceptable When dual-system operating cost is prohibitive Data mesh + platform enablement Domain teams Scrum; platform team enables via embedded squads Scaling Agile across multiple domain teams; federated ownership When platform team is undersized relative to domain team count

4.1 Pure Scrum (Software-Derived)

A. Pure Scrum (Software-Derived)

The first pattern is the direct adoption of software Scrum without modification. The team uses two-week sprints with story-point estimation, sprint goals, daily standups, sprint reviews, and retrospectives. All work items, whether feature development, infrastructure work, or bug fixes, go into a single sprint backlog and are estimated against the same scale.

Pure Scrum works in data engineering contexts where the team is small (typically three to five people), the requirements come from a single product owner or stakeholder group, and the underlying infrastructure is stable. Analytics teams that build narrowly scoped dashboards for a single business unit are the most reliable application of pure Scrum in the author's observation. Where these conditions hold, the standard Scrum mechanics produce predictable cadence and clear progress.

Pure Scrum fails reliably in data engineering contexts where the team has cross-team dependencies, mixed exploratory and delivery work, or significant infrastructure churn. The earliest Scrum attempts at the case study site, between 2012 and 2015, illustrated these failures. Story points for infrastructure work were wildly unreliable: a Kubernetes cluster setup task estimated at 8 points consumed 40 actual points across three sprints because each step uncovered prerequisites that had not been visible at planning time. Sprint plans became fantasy documents that the team adjusted informally throughout the sprint, and retrospectives accumulated complaints about estimation accuracy that no estimation refinement could actually fix because the underlying problem was not estimation but the unpredictability of infrastructure work itself.

Practitioner observation: pure Scrum is appropriate in data engineering contexts that resemble software engineering contexts. It is not the right starting point for teams whose work routinely involves infrastructure or cross-team dependencies, and trying to make it work in those contexts produces the kind of organizational friction that drives Agile cynicism.

B. Scrum and Kanban Hybrid

The hybrid pattern decouples predictable from unpredictable work by assigning each to a different methodology. Domain teams use Scrum for feature delivery: two-week sprints, sprint goals, story-point estimation, the standard ceremonies. The platform team uses Kanban for infrastructure work: requests flow continuously into a board, are prioritized by impact and urgency, and are pulled by available engineers without sprint boundaries. The two methodologies are bridged by explicit service level agreements (SLAs) that define how quickly the platform team commits to provisioning common infrastructure requests. Domain teams plan their sprints with knowledge of these SLAs and anticipate infrastructure needs one or two sprints in advance.

The hybrid works because it acknowledges that infrastructure work has properties (unpredictable scope, externally driven priorities, frequent emergencies) that resist time-boxing, while feature delivery work has properties (bounded scope, internal prioritization, predictable cadence) that benefit from it. Forcing both into the same methodology produces poor outcomes for both. Separating them allows each to operate under conditions that fit its actual nature.

The hybrid has its own challenges. The platform team can become a bottleneck if the request volume exceeds its capacity, and the failure mode is silent: domain teams continue to file requests, the queue grows, and the SLAs that the platform team committed to begin to slip without anyone explicitly acknowledging the slippage until a downstream sprint goal fails. Mitigations include WIP limits on the platform Kanban board, explicit escalation paths when SLAs are at risk, and the platform enablement squad pattern discussed in Section 4.5. The case study site adopted the hybrid as its default configuration after the Sprint 7 failure described in Section 6, and the configuration produced sprint goal completion rates in the 82 to 85

percent range across domain teams against a baseline of approximately 60 to 65 percent under pure Scrum.

C. Narrative Sprints

The narrative sprints pattern shifts the unit of commitment from story points to outcome-framed sprint goals. Rather than committing to a particular number of story points, the team commits to outcomes such as \"ship daily revenue dashboard,\" \"reduce query latency by 55 percent,\" or \"enable self-service reporting for the CFO team.\" Stories within the sprint are framed as user-oriented narratives: \"as an analyst, I want to filter by region so that I can analyze regional trends.\" Acceptance criteria describe outcomes rather than tasks, and scope can flex within the sprint goal as long as the goal itself remains achievable.

Narrative sprints work well for cross-functional teams where the deliverable is a data product whose value depends on outcomes rather than on specific tasks. They work less well in contexts that require story-point-level forecasting for external commitments, because they intentionally weaken the link between effort estimation and time prediction. The case study site adopted narrative sprint goals for analytics teams whose stakeholders cared about outcomes, while retaining story-point tracking for engineering teams whose stakeholders required forecasting against external commitments.

D. Strangler Fig Migration

The strangler fig pattern, named by Fowler (2004) after the strangler fig vine that gradually envelops and replaces its host tree, addresses large-scale system migration without flag-day cutover risk. Old and new systems run in parallel; traffic is incrementally redirected from the old to the new; the old system continues to operate until confidence in the new system is high enough to retire it. The pattern allows Agile iteration on the new system without breaking the workloads that depend on the old.

The case study site used the strangler fig pattern for the migration from a Hadoop-era enterprise data warehouse to an on-premises lakehouse running on Apache Iceberg and Trino over a Ceph storage layer. The legacy ETL pipelines continued to operate while new Airflow and Iceberg pipelines ran in parallel. Traffic was redirected domain by domain: the chief financial officer's dashboard moved first because its data was stable enough for confident validation, then non-critical sources, then high-volume transactional data. Each domain redirection became a sprint goal in its own right (\"redirect the CFO dashboard to the lakehouse\"), which made the migration amenable to incremental Agile delivery rather than requiring a multi-quarter waterfall plan.

The challenges of the strangler fig pattern are well known. Operating dual systems imposes operational overhead that compounds over the migration period. Code duplication between old and new systems is sometimes unavoidable. The last 10 percent of the migration tends to take as long as the first 90 percent, because it includes the cases that resisted migration earlier and that often have the most complex dependencies. The case study experience matched these expectations; the migration ran an estimated 30 months, with the final transactional data sources requiring substantially more effort per source than the early non-critical sources had.

E. Data Mesh with Platform Enablement

The fifth pattern combines a federated organizational structure (domain teams own their data products and pipelines) with a platform team that provides shared infrastructure and tools. The architectural framing is data mesh (Dehghani, 2019), but the Agile implications are distinct from the architectural ones. Domain teams operate on two-week Scrum sprints. The platform team operates on quarterly planning cycles, with explicit alignment sessions in which domain teams articulate the platform features they will need over the coming quarter and the platform team commits to delivery. Platform engineers are also embedded with domain teams, in the squad pattern discussed in Section 7, to reduce the latency of small infrastructure requests below the cadence of the platform team's quarterly cycle.

The pattern scales Agile across multiple teams by acknowledging that not all teams need to operate on the same cadence. Domain teams benefit from the predictable rhythm of two-week sprints; the platform team benefits from longer planning horizons that absorb the uncertainty of larger infrastructure work. Quarterly alignment prevents the misalignments that would otherwise emerge between teams operating on different cadences.

The case study site converged on this pattern after several years of iteration. The restructuring from eight direct reports to five (covering engineering, analytics, machine learning, platform, and governance) was part of the transition, removing structural redundancy that had accumulated under earlier configurations and aligning team boundaries with the data mesh model of domain ownership plus platform enablement. The outcome metrics are reported in Section 7.

Practitioner observation: the choice among these five patterns is principally a function of team size, infrastructure stability, and the organization's tolerance for separate methodologies running in parallel. Small teams with stable infrastructure can use pure Scrum. Larger teams or teams with unstable infrastructure benefit from the hybrid or from the data mesh pattern. The strangler fig is a tactical pattern for migration projects rather than an organizing methodology, and it can coexist with any of the others. Teams that try to enforce a single methodology across heterogeneous work tend to find that the methodology produces friction in the parts of the work it does not fit.

V. CHALLENGES UNIQUE TO DATA ENGINEERING

==========

This section addresses RQ2 by examining five challenges that data engineering teams encounter when adopting Agile,

and the adaptations that have proved effective in addressing them. The challenges are not exhaustive, and they are not all unique to data engineering in an absolute sense; they recur in data engineering with a frequency and severity that distinguish them from typical software engineering Agile adoption.

5.1 Estimation for Unknown Unknowns

A. Estimation for Unknown Unknowns

Story-point estimation assumes that the work to be estimated has bounded scope and that the principal uncertainty is in the effort required to complete known tasks. In data engineering, the unknown unknowns are often more consequential than the effort estimation. A story to ingest a new API may be estimated at 5 points based on the assumption that the API behaves as documented; during the sprint, the team discovers that the API is rate-limited at a level lower than the consumption rate, that the schema is inconsistent across endpoints, and that the API owner does not respond to questions in any reasonable timeframe. The story ends as a 30-point investment whose original 5-point estimate was not wrong about the coding work; the estimate failed because it did not account for the scope discovery that the work itself produced.

The Sprint 7 narrative described in Section 6 illustrates the pattern at scale. A Kafka topic migration was estimated at 13 points on the basis of \"copy topics and update consumers.\" During execution, the team discovered that Kafka rebalancing on the largest topic required 36 hours instead of the expected 4, that consumer code required refactoring for the new cluster's configuration, and that legacy configuration incompatibilities produced approximately 0.1 percent message loss that required investigation. The original estimate was not the cause of the failure; the cause was that the estimation framework assumed bounded scope where unbounded scope was the actual condition.

The academic literature on estimation accuracy in Agile is uneven. Software engineering estimation studies including Cohn (2005) and Grenning (2002) document that story points are poor predictors of calendar time even in software contexts. Empirical work on estimation accuracy in data engineering specifically is sparse, and the available evidence (drawn primarily from practitioner reports rather than peer-reviewed studies) suggests that the gap between estimate and actual is wider than in software engineering, plausibly because the proportion of unknown unknowns is higher.

Four adaptations have proved effective at the case study site. Range estimation replaces single-point estimates with low-likely-high triples (an API ingest might be estimated at 3-5-13 points), which forces explicit acknowledgement of uncertainty without producing pessimism in nominal estimates. Time-boxed exploration explicitly schedules investigation of uncertain work before commitment: a 3-day exploration of an API can uncover its rate limits and schema characteristics before the team commits to an integration estimate. Infrastructure separated as a queue removes the most variable work from the estimated sprint backlog and tracks it through Kanban with WIP limits. Estimation retrospectives, conducted monthly rather than at every sprint review, surface patterns of over- and under-estimation by work type and inform future planning.

Outcome at the case study site: estimation accuracy improved from an estimated ±55 percent to approximately ±25 percent on subsequent sprints, and sprint goal completion rates rose from approximately 60 percent to 75 percent in the immediate post-Sprint-7 period.

5.2 Infrastructure Dependencies and Cross-Team Blocking

B. Infrastructure Dependencies and Cross-Team Blocking

Software engineering Agile assumes low cross-team coupling. Data engineering routinely violates this assumption. A domain team's sprint depends on the platform team's infrastructure (a new Kafka cluster, additional Kubernetes capacity, a schema registry update); if the platform team slips, the domain team is blocked. The blocking is not ordinary delay; it cascades, because the dependent work cannot proceed and the team's velocity collapses for the duration of the block.

Project management methodologies designed for high-dependency environments (critical path analysis, program evaluation and review technique) provide vocabulary for thinking about cross-team blocking, but they sit uncomfortably with Agile assumptions of team autonomy. The literature on Agile coordination across high-dependency teams is comparatively thin, and most of the available guidance comes from large-scale Agile frameworks (Scaled Agile Framework, Disciplined Agile Delivery, Large-Scale Scrum) whose treatment of cross-team dependencies is more procedural than empirical.

Four adaptations have proved effective. Explicit dependency mapping at quarterly planning sessions identifies which domain teams depend on which platform features, allowing the platform team to sequence its work to unblock critical paths. Platform enablement squads, the subject of the case study in Section 7, embed platform engineers with domain teams to handle small infrastructure requests synchronously rather than queuing them through the platform team's planning process. Dependency SLAs commit the platform team to specific provisioning latencies (a new Kafka topic within 3 business days, a new Trino catalog within 5 business days) that domain teams can plan against. Risk backlogs maintained alongside the sprint backlog explicitly track infrastructure risks that might block the sprint, with mitigation plans identified before the sprint begins.

Outcome at the case study site: infrastructure blocking fell from about 30 percent of sprints (where domain teams spent meaningful time waiting) to less than 5 percent. The

case study in Section 7 reports the platform enablement squad outcomes in detail.

C. Exploratory Versus Delivery Work

Data engineering teams routinely combine two kinds of work that resist combination. Delivery work (build the daily refresh of the customer 360 view, integrate the new supplier feed) has bounded scope and can be planned and executed within sprint boundaries. Exploratory work (figure out which features predict customer churn, investigate why query performance has degraded) has unbounded scope and resists time-boxing in the same way. Mixing the two in a single sprint produces consistent friction, because either the exploratory work absorbs more time than planned at the expense of the delivery work, or the delivery work is prioritized and the exploratory work is starved.

The case study site experienced this directly when an analytics team sprint included both \"build customer 360 dashboard\" (delivery, scoped) and \"explore which features predict lifetime value\" (exploratory, uncertain). The exploratory work consumed 60 percent of the sprint's available time before being deprioritized as the dashboard deadline approached, and the deprioritization felt to the team like punishment for doing the exploratory work in good faith. The team retrospective concluded that the mixture, not either kind of work alone, had produced the friction.

Four adaptations have proved effective. Separation of exploration from delivery places exploratory work in a separate backlog or under the ownership of a separate sub-team, allowing each kind of work to operate under conditions appropriate to its nature. Time-boxed exploration imposes explicit limits on exploratory work (a 2-week exploration on churn prediction, then commit to build or abandon) while preserving the openness of the exploration within the box. Ratio-based planning caps exploratory work at 20 to 30 percent of sprint capacity, ensuring that delivery work has priority without eliminating innovation. Parallel streams, feasible only in larger teams, dedicate one or two engineers to exploration on a continuous basis while the rest of the team operates on delivery sprints. The case study site implemented the separation pattern: a 2-person continuous exploratory squad operated alongside a 4-person delivery team, and exploratory discoveries that proved promising were converted to delivery stories with the additional information that the exploration had produced.

Outcome: the delivery team achieved roughly 85 percent sprint goal completion after the separation, against the 60 to 65 percent baseline that had prevailed when exploratory and delivery work were mixed. The exploratory squad produced several discoveries that fed into subsequent delivery sprints, and the conversion rate from exploration to production work became a useful metric in its own right.

D. Pipeline Quality and Late-Appearing Failures

Software engineering Agile relies on automated tests as the primary signal of done-ness: a feature is done when its tests pass. Data engineering quality failures often surface long after deployment. A pipeline that loads data successfully on day 5 of a sprint may produce subtly wrong data that is only noticed weeks later when an analyst spots a discrepancy or when a downstream consumer derives an incorrect aggregate. The sprint review at the end of the sprint cannot reliably evaluate whether the data product is actually working, because the failure surface has not yet occurred.

The implication for the definition of done is direct. Delivery of a pipeline is not the same as quality assurance of a pipeline. The case study site experienced this in the aftermath of Sprint 7: a pipeline shipped on time for a different work item, and an estimated two weeks later a quality check revealed that 5 percent of records had a null customer_id field that should have been caught before deployment. The remediation work distracted the next sprint and produced exactly the kind of compounding loss that Agile is supposed to prevent.

Four adaptations have proved effective. Data quality as an acceptance criterion includes specific quality metrics in sprint acceptance (a pipeline must achieve 99 percent completeness for customer_id) that are verified before sprint closure. Post-sprint SLA monitoring tracks deployed pipelines for one to two sprints after delivery, with any quality violations treated as urgent and triggering post-sprint bug fixes that are explicitly accounted for in subsequent sprint capacity. Data contracts formalize quality agreements between producers and consumers and are enforced automatically at deployment through tools such as Great Expectations. Continuous quality monitoring runs Great Expectations and anomaly detection 24/7 rather than only at sprint boundaries, alerting on deviations from baseline as they occur.

Outcome: integrating Great Expectations into the Airflow DAGs at the case study site, with deployment blocked on expectation failure, caught three anomalies in the first month after introduction that would otherwise have produced downstream impact. The definition of done expanded to include quality verification, and the change in practice was notable in subsequent sprint reviews: teams could state with confidence that delivered pipelines were not just deployed but actually working.

E. Estimation Uncertainty and Commitment Forecasting

Story-point velocity is sometimes presented as a forecasting tool: if a team's velocity averages 40 points per sprint, then a 120-point feature should ship in three sprints. The forecasting interpretation is hazardous in software engineering and worse in data engineering. Velocity is a measure of past throughput under particular conditions, and projecting it forward assumes that future conditions will resemble past conditions, which in data engineering they often do not.

The case study site experienced the forecasting failure directly. The CFO asked when the revenue dashboard would refresh daily; the team estimated 20 story points and committed to delivery within three sprints. Sprint 1 delivered 18 points (velocity on plan). Sprint 2 delivered 8 points because a Spark version upgrade introduced infrastructure work that the original estimate had not contemplated. Sprint 3 resumed feature delivery and added 15 points. The total was 41 points across more than three sprints, against the 20-point original estimate. The forecast had been overconfident not because the team was incompetent at estimation but because the underlying conditions changed in ways that the planning had not anticipated.

Four adaptations have proved effective. Cycle time tracking measures the calendar duration from story creation to completion as the primary forecasting input, replacing velocity as the predictor of calendar time. Monte Carlo forecasting uses past cycle-time distributions to simulate possible delivery scenarios and reports probability ranges (\"90 percent confidence the feature ships in 3 to 5 sprints\") rather than point estimates. Commitment forecasting commits to outcomes by a calendar date (\"the revenue dashboard will refresh daily by the end of the quarter\") rather than to specific sprint delivery, allowing flexibility within the larger window. Frequent re-planning re-forecasts every two to three sprints based on new cycle-time data and risk discovery rather than treating the original forecast as a fixed plan.

Outcome: the case study site shifted from sprint-level commitments to quarterly outcome commitments and from velocity-based forecasts to probability ranges. The CFO began receiving probability statements (\"90 percent chance the revenue dashboard ships by Q3, 70 percent chance by mid-Q3\") instead of promises, and the accuracy of quarterly commitments improved to approximately 80 percent while team satisfaction rose because the planning conversations stopped requiring false certainty.

VI. CASE STUDY: SPRINT 7 FAILURE AND RECOVERY

Week 2 was different. The Kafka rebalancing process for the largest topic, a 500GB trading data topic, took 36 hours instead of the expected 4. The cause was that the topic's partition layout required careful sequencing to avoid message loss during rebalancing, and the team had not characterized the topic in advance because the rebalancing tooling was supposed to handle the variation automatically. The investigation consumed an estimated 8 unplanned story points and was added to the sprint backlog. Concurrently, the API integration team discovered that the first provider's API was rate-limited at 100 requests per minute against the team's assumed 200, requiring a caching component that had not been planned. That work was estimated at 5 unplanned points and added to the sprint. Late in week 2, post-migration validation on the Kafka work revealed 0.1 percent message loss, attributable to a legacy Kafka configuration that was incompatible with the new cluster. The investigation of the message loss was estimated at 5 additional unplanned points and added to the sprint. The cumulative scope was now 68 points, against an initial commitment of 55.

Week 3, which existed because the sprint extended slightly past its nominal boundary, was the worst of the sprint. The Kafka message loss investigation began to block the API team, who had planned to use Kafka as a buffering layer for the API ingestion and could not safely do so until the message loss was understood and addressed. The team converged on Kafka debugging, with all available engineers working on the investigation, and the API work stalled. An estimated 8 additional unplanned points were added for the infrastructure troubleshooting. The total sprint scope was now 76 points, more than 38 percent above the original commitment.

The sprint review on day 15 was painful. The team had completed nearly 32 of the 76 attempted points, which was 42 percent of the attempted scope and approximately 58 percent of the original commitment. The sprint goal had failed: neither the Kafka migration nor the API integration had reached a state that the stakeholders considered done. The retrospective that followed was honest and unstructured. Team members were demoralized, defensive, and tired. The retrospective produced several observations that subsequently informed the recovery.

6.3 Retrospective Findings

A. Context and Sprint Goal

The team consisted of nearly eight data engineers split between platform and domain work. Sprint cadence was two weeks, with sprint planning on Monday morning of week 1 and sprint review and retrospective on Friday afternoon of week 2. Velocity over Sprints 1 through 6 had averaged 35 to 38 story points, and the team had become comfortable committing in the 35 to 40 point range with an acceptable goal completion rate.

Sprint 7 broke from the established pattern. Domain stakeholders (the CFO, the machine learning lead, and the analytics lead) had identified upcoming-quarter needs that included new data sources requiring integration through APIs from three external providers. Independently, the platform team had identified the need for a Kafka cluster upgrade including rebalancing and capacity expansion, which was overdue and which had been deferred from earlier sprints. The Sprint 7 planning meeting decided to do both pieces of work in the same sprint, on the implicit theory that the team could partition itself (four engineers on the Kafka migration, four on the API integration) and that each work stream could proceed in parallel without significant interaction.

The sprint commitment was 55 story points: 20 points for the Kafka migration (\"copy old topics to the new cluster, validate messages, redirect consumers\") and 35 points for

the API integration work (\"build the ingestion framework, integrate the three providers\"). No explicit risk flags were raised at planning. In retrospect, the absence of risk flags was itself a warning sign: a sprint that combined an infrastructure migration and three external integrations should have produced a long list of identified risks, and the absence of the list reflected the team's inexperience with risk identification rather than the actual absence of risks.

B. The Sprint Itself

Table 2 summarizes the events of the sprint by week, and the discussion that follows treats them in narrative form.

Table 2. Sprint 7 timeline.

Period **Phase** **Events** **Cum. Scope** ----- -------- -------------- --- -- -- -- -------------------------------- ---------------- Days 1--5 Week 1 Kafka migration begins; API ingestion framework underway. Velocity tracking on plan; no surprises surfaced. 55 pts Days 6--10 Week 2 Kafka rebalancing on the 500GB trading topic took 36 hours instead of the estimated 4. First API discovered to be rate-limited at 100 req/min against an assumed 200 req/min, requiring a caching component. Post-migration validation revealed 0.1% message loss from a legacy config incompatibility. 68 pts Days 11--15 Week 3 Kafka message loss investigation blocked the API team, who had planned to rely on Kafka for buffering. The team converged on Kafka debugging, leaving API work stalled. No formal mid-sprint replan occurred; scope continued to grow informally. 76 pts Day 15 Sprint review 32 of 76 attempted points completed (42% of attempted scope; 91% of original commitment in raw points but the goal itself was missed). Sprint goal failed.

Week 1 began on plan. The Kafka migration team began copying topics from the old cluster to the new, completing approximately 10 of the 20 estimated points by Wednesday. The API integration team built the ingestion framework and began the first provider's integration, completing 8 points by the same point. The team's daily standup on Wednesday recorded a sprint that was tracking to commitment, and there was no indication that anything was about to go wrong.

C. Retrospective Findings

The first retrospective finding was that the root cause of the failure was over-commitment rather than estimation error. Even if the team had estimated the work at 76 points instead of 55, the sprint would still have failed, because the unknowns that emerged during execution could not have been estimated in advance. The estimation conversation was therefore a distraction from the more important conversation about commitment volume and risk identification.

The second finding was that risk identification at planning had been inadequate. The Kafka rebalancing risks were not articulated. The API rate limits were not investigated. The interactions between the two work streams were not anticipated. None of these were unknowable; all of them could have been surfaced through a more deliberate planning process that explicitly asked what could go wrong and what the team would do about it if it did.

The third finding was that mid-sprint replanning had happened informally but had not been formalized. As the sprint deteriorated, the team had shifted resources, deferred some work items, and converged on the most critical path, but no formal mid-sprint replan had occurred and no stakeholder communication had explained the changes. The team had done the right operational work without the right organizational acknowledgment, and the absence of the acknowledgment was part of why the retrospective felt so painful: the failure could have been framed as a transparent course correction rather than as a hidden disaster.

The fourth finding was that the API rate-limit discovery could have been caught in a one-day pre-sprint investigation of each API. The investigation had not been scheduled because the team had assumed the APIs behaved as documented. The assumption was specifically the kind of assumption that a deliberate risk-identification process would have flagged for verification.

The fifth finding was that knowledge silos had contributed to the cascade. Only one engineer had detailed knowledge of the Kafka cluster topology, and that engineer became the bottleneck for the rebalancing investigation. The cross-training that would have allowed multiple engineers to contribute to the investigation had not been done.

D. Recovery and Adaptation

The recovery from Sprint 7 took the form of a focused one-week planning session for Sprint 8, in which the team revisited its commitment process, its risk identification practices, and its mid-sprint mechanics. Sprint 8 committed to 38 story points, a deliberate scaling back from the 55 of Sprint 7 and a return to the 35 to 38 range that had been comfortable in earlier sprints. The Kafka migration was split into three separate sprints rather than attempted as a single effort. Two of the three APIs were deferred so that Sprint 8 focused on integrating only one. An explicit risk identification step was added to sprint planning, in which the team identified the top three risks per story and proposed a mitigation plan for each. Mid-sprint checkpoints were introduced on the Wednesday of week 2 of each sprint, with a go or no-go decision on whether the sprint goal was still achievable and a formal replan if not. Knowledge-sharing sessions were scheduled to cross-train the team on critical infrastructure components, beginning with the Kafka topology that had bottlenecked Sprint 7.

Sprint 8 outcomes: the team committed to 38 points and completed 36, for an around 94 percent goal completion rate. Team morale improved noticeably; the retrospective reported that Sprint 8 felt sustainable in a way that Sprint 7 had not, and that the mid-sprint transparency made the work feel less like a fire drill. Subsequent sprints (Sprints 9

through approximately 20) consistently achieved 80 to 90 percent goal completion against commitments in the 35 to 40 point range, against the pre-Sprint-7 baseline of 65 percent.

Beyond the immediate sprint mechanics, Sprint 7 catalyzed several broader organizational changes that took effect over the following months. The platform enablement squad model described in Section 7 was conceived in response to the infrastructure-blocking pattern that Sprint 7 illustrated. The strangler fig pattern was adopted as the default approach for large migrations, on the explicit theory that any migration that could be done in a single sprint should not be done that way. Quarterly planning was introduced to align domain team roadmaps with platform team capacity. Range estimation replaced single-point estimation for stories with significant uncertainty. The risk backlog became a standing artifact alongside the sprint backlog. The mid-sprint checkpoint became a permanent fixture in every sprint, not only the ones that appeared to be in trouble.

E. Lessons

The principal lesson from Sprint 7, in the author's reflection, is that data engineering work has irreducible uncertainty that no estimation refinement can eliminate. The framing of estimation as the central problem in Agile data engineering is a category error: the central problem is commitment under uncertainty, and the workable response is to commit less, identify risks more systematically, and provide mechanisms for honest mid-sprint replanning when the inevitable surprises emerge. Teams that try to estimate their way to better outcomes will continue to encounter the kinds of failures that Sprint 7 represents; teams that adopt the mechanisms of risk identification, range estimation, and mid-sprint adaptation will encounter them less often and will recover from them more quickly when they do.

A secondary lesson is that the painfulness of a failed sprint is partly avoidable through transparency. The Sprint 7 retrospective was painful because the failure had been hidden inside the sprint until the end, and the stakeholders learned about it only at the sprint review. Subsequent sprints, with mid-sprint checkpoints and explicit communication when goals were at risk, produced smaller surprises and more constructive conversations even in cases where the sprint goal was ultimately not achieved. The information was not less unwelcome; the timing of its delivery made it actionable rather than merely demoralizing.

VII. CASE STUDY: PLATFORM ENABLEMENT SQUAD AND THE 20 PERCENT EFFICIENCY GAIN

The expected outcome was that infrastructure latency for small requests would fall from 2 to 3 weeks (the typical interval under the prior async model) to approximately 1 to 3 days, and that the reduction would translate into improved sprint goal completion by removing the principal cause of mid-sprint surprise.

7.3 Pilot: Analytics Domain Team, Q1 2019

A. Context

By early 2019, the case study site comprised close to 30 data professionals organized under five direct reports following the restructuring described in Section 4.5: engineering, analytics, machine learning, platform, and governance. Domain teams (engineering, analytics, ML) operated on two-week Scrum sprints. The platform team operated on Kanban with quarterly planning. Despite the methodological hybrid, domain teams continued to experience infrastructure blocking with sufficient frequency to suppress velocity. Sprint goal completion across domain teams averaged nearly 60 to 70 percent, with the principal cause of incomplete goals being infrastructure dependencies that the platform team had not delivered on the timeline that the domain team had assumed.

The platform team was not idle. It was managing a portfolio of more than 1,000 infrastructure items spanning Kubernetes nodes, Ceph clusters, Purview policies, Collibra access control configurations, and the recurring operational work of running a production lakehouse. Small infrastructure requests (typically one to two days of work to provision a new Kafka topic, add a Trino catalog, or update a governance policy) were handled when capacity allowed, but were not formally tracked through the sprint backlog because the overhead of formalization exceeded the work itself. The result was that small requests accumulated in informal channels, took longer than they should have, and emerged as sprint blockers when the requesting domain team needed them.

B. The Squad Concept

The platform enablement squad concept was simple: embed one platform engineer with each domain team, on a continuing basis, to act as the on-call infrastructure engineer for that team. The embedded engineer would attend the domain team's standups, sprint planning, and retrospectives; would triage infrastructure requests as they emerged; would make go or no-go decisions on prioritization against the platform team's broader work; and would either execute small requests synchronously (within one to two days) or escalate larger ones to the platform team. The model is consistent with the enabling team pattern described by Skelton and Pais (2019) in the Team Topologies framework, in which a team of specialists is deployed to accelerate the work of stream-aligned teams without taking ownership of the stream-aligned teams' deliverables.

C. Pilot: Analytics Domain Team, Q1 2019

The pilot deployed one platform engineer with the analytics domain team for a three-month period beginning in Q1 2019. The pilot was deliberately scoped to a single domain team to allow careful observation of the effects without introducing the confounding variation of multiple simultaneous deployments. The analytics team was selected

because its sprints had the most consistent pattern of infrastructure-blocked velocity loss and because its lead was supportive of the experiment.

The quantitative outcomes over the three-month pilot period were the following. Infrastructure blockers experienced by the analytics team fell from approximately 8 per quarter (each lasting 2 to 3 weeks) to about 2 per quarter (each lasting 2 to 5 days). Sprint velocity rose from approximately 70 story points per sprint to nearly 82, an increase of an estimated 17 percent. Sprint goal completion rose from approximately 60 percent to 85 percent, an increase of 25 percentage points. The retrospective feedback from the analytics team was succinct: infrastructure was no longer a surprise, it was planned and de-risked.

The pilot's success was not a surprise in direction; it was a surprise in magnitude. The team had expected a measurable improvement, but the magnitude of the velocity gain and the goal completion improvement exceeded the planning assumptions and provided the empirical basis for the broader rollout.

D. Full Rollout: Q2 through Q4 2019

The full rollout extended the pattern to the engineering and machine learning domain teams over Q2 through Q4 2019, ultimately deploying three embedded platform engineers across three domain teams. Each embedded engineer maintained a parallel relationship with the platform team through a weekly sync that allowed cross-team de-duplication of work, sharing of solutions to recurring problems, and escalation of issues that required broader attention.

Table 3 summarizes the outcomes of the full rollout against the pre-squad baseline.

Table 3. Platform enablement squad outcomes (12 months full rollout).

Metric **Pre-Squad Baseline** **Post-Squad (12 months)** ---------- ---------- ---------- Quarterly velocity (story points) 200 250 (+20%) Sprint goal completion rate 65% 82--85% Infrastructure latency (typical request) 2--3 weeks 3--4 days Infrastructure blockers per quarter (analytics pilot) 8 blockers (2--3 weeks each) 2 blockers (2--5 days each) Quarterly infrastructure requests handled Limited by sprint cadence 150+ requests

The headline outcome is the 20 percent overall efficiency gain in domain team velocity (from an estimated 200 quarterly story points to nearly 250) with the corresponding rise in sprint goal completion from an estimated 65 percent to around 82 percent. The 150 or more infrastructure requests per quarter handled at an average latency of 3 to 4 days represented a substantial increase in throughput against the prior arrangement, in which the same volume of requests would have been handled at much higher latencies and would have produced correspondingly more sprint blockers.

A secondary outcome that the team did not initially anticipate was that the platform team itself benefited from the squad model. With the small reactive requests absorbed by the embedded engineers, the platform team's central capacity was freed for strategic infrastructure work that had been deferred under the prior arrangement. Over the rollout period, the platform team completed three major initiatives (a Kubernetes upgrade, a Ceph capacity expansion, and the Purview governance framework rollout) that had been on the platform roadmap for more than a year without progress. The squad model thus produced gains on both sides of the embedded relationship: domain teams shipped more, and the platform team made progress on the strategic work that had been crowded out by the constant pressure of small reactive requests.

E. Economics

The cost of the squad model was close to \$300,000 per year, calculated as the loaded cost (salary plus benefits) of three full-time-equivalent embedded engineers. The benefit, calculated as the opportunity cost of the productivity that would have been lost without the squad, was close to equal: across fifteen domain team engineers earning average salaries, a 20 percent productivity gain represents approximately \$300,000 in reclaimed value annually. The first-year ROI was therefore approximately cost-neutral.

The cost-neutral first-year framing is the conservative reading. Several considerations make the actual return higher in subsequent years. The platform team's strategic initiatives, completed because of the capacity freed by the squad model, would have produced their own value that the cost calculation does not directly capture. The improvement in domain team morale, retention, and predictability of delivery has its own value that resists straightforward quantification. The compounding effect of better sprint outcomes (better forecasts allow more confident commitments to stakeholders, which produces better business outcomes, which produces better feedback loops) accumulates over time. By year two, the program was unambiguously net-positive, and the budget conversations about its continuation were straightforward in a way that the initial budget conversations about its launch had not been.

F. Operational Model

The operational model that emerged from the rollout had several features that the planning had not fully specified in advance and that subsequent teams adopting the pattern may want to consider.

The embedded engineer's responsibilities centered on the on-call presence with the domain team: attendance at standups, sprint planning, and retrospectives; triage of infrastructure requests as they emerged; go or no-go decisions on prioritization against the platform team's broader work; synchronous execution of small requests (under close to 2 days of work) and escalation of larger ones. Backlog management was handled through a separate column on the domain team's Kanban board, with priorities driven by impact and urgency; the embedded engineer's

work was tracked separately from the domain team's sprint velocity to avoid distorting the velocity metric.

Communication between the embedded engineer and the platform team happened through a weekly sync in which embedded engineers shared what they were seeing, escalated blockers, and de-duplicated work that was emerging in multiple domain teams simultaneously. Knowledge sharing was deliberate: embedded engineers documented their solutions in runbooks and FAQs that the domain team could refer to, and a monthly cross-team session allowed embedded engineers to present learnings to other domain teams. The knowledge-sharing function turned out to be one of the unexpected benefits of the model: the embedded engineers became natural conduits for cross-team learning that had been difficult to achieve under the prior arrangement.

G. Lessons and Caveats

Five lessons emerged from the program that the author offers as practitioner observations. First, embedded engineers are a force multiplier for domain team velocity in environments where infrastructure blocking is the dominant cause of velocity loss. The 20 percent gain at the case study site is not universally portable; it reflects the specific baseline (60 to 65 percent goal completion) and the specific causes of that baseline. Teams whose velocity is suppressed by other factors (estimation issues, scope unclarity, organizational misalignment) should not expect the same gains from the same intervention.

Second, the role of the embedded engineer requires careful definition. The embedded engineer is an infrastructure specialist, not a domain specialist, and cannot reasonably be expected to take ownership of domain knowledge. The temptation to ask the embedded engineer to own progressively more of the domain team's work must be resisted; doing so erodes the benefits of the squad model and creates dependency relationships that the platform team cannot sustain.

Third, burnout risk is real. The embedded engineer can be overloaded if infrastructure requests exceed capacity, and the on-call posture creates a feeling of constant availability that is difficult to sustain over time. Backpressure mechanisms (escalation to the platform team for work that exceeds capacity, explicit time off, rotation through the embedded role) are necessary for the long-term health of the engineers.

Fourth, knowledge transfer is one of the unexpected benefits. The embedded model is the most effective vehicle for cross-team learning that the case study site found, because it places engineers in direct contact with each other's constraints and produces shared language and empathy that other forms of cross-team interaction did not.

Fifth, scalability has limits. The model worked for three to four domain teams at the case study site. Beyond that scale, the platform team became too stretched to support both the embedded engineers and its own central work, and the model would have required either revisiting the approach or expanding the platform team headcount. Organizations contemplating the pattern should plan for the scale at which they expect to operate and budget accordingly; the pattern is not free, and pretending that it is will produce the same kinds of overload that the pattern is intended to address.

VIII. DISCUSSION AND FUTURE RESEARCH DIRECTIONS

==========

The findings from the survey and the two case studies together support a synthesis of the current state of Agile in data engineering and identify open challenges that the literature has not yet addressed in depth.

8.1 Synthesis

A. Synthesis

Agile works for data engineering teams, but the patterns that work are not the patterns that work for software engineering teams. Pure Scrum imported from software contexts produces the kinds of failures that Sprint 7 illustrated, and the failures are not edge cases; they are predictable consequences of the friction between software Agile assumptions and data engineering conditions. The patterns that work in data engineering are hybrid (Scrum for predictable feature work, Kanban for unpredictable infrastructure work), federated (domain teams own their data products with platform team enablement), and explicit about the limits of estimation under uncertainty (range estimation, time-boxed exploration, quarterly outcome commitments rather than sprint-level forecasts).

The challenges that data engineering teams face are not all unique to data engineering in an absolute sense, but they recur with a frequency and severity that distinguish data engineering from typical software engineering Agile contexts. Estimation for unknown unknowns, infrastructure dependencies, exploratory versus delivery work, late-appearing pipeline failures, and forecasting under uncertainty all have software engineering analogs, but in data engineering they are the rule rather than the exception, and they require deliberate adaptation rather than ad hoc handling.

The measurable outcomes from the case study site are consistent with the literature on Agile productivity in software engineering, scaled appropriately for the data engineering context. The 20 percent efficiency gain from the platform enablement squad initiative is in the range of what large-scale Agile transformations report (Dikert, Paasivaara, and Lassenius, 2016). The 82 to 85 percent sprint goal completion rates achieved after adaptation are at the high end of what well-functioning Agile teams typically achieve. The improvements are real, and they are achievable through the kinds of adaptations the case study describes, but they

require sustained organizational commitment rather than tooling or training alone.

B. Open Challenges

8.2.1 Estimation for Exploratory Work

No clear consensus exists on how to estimate exploratory data science or machine learning work. Time-boxing imposes structure on exploration without addressing the underlying uncertainty about whether the exploration will produce useful results. The case study site separated exploration from delivery into different work streams, which addressed the friction but did not solve the estimation problem itself; the exploratory squad operated continuously without committing to specific outcomes, and the cost was that its work could not be easily forecast for stakeholders. Research on exploratory work estimation, particularly in machine learning contexts, remains an open area.

8.2.2 Cross-Organizational Agile Coordination

Data mesh and similar federated patterns extend across organizational boundaries: a domain team's data product may depend on partner APIs, vendor data feeds, or upstream systems owned by entirely separate organizations whose release schedules and quality commitments are not under the consuming team's control. Agile coordination across these boundaries is not well addressed by the existing literature, which generally assumes that Agile teams can be planned within a single organizational structure.

8.2.3 Agile for Infrastructure-Heavy Work

Kubernetes deployments, Ceph storage expansions, and similar infrastructure work have constraints that pure Agile does not handle well: hardware lead times, compatibility testing matrices, capacity planning that must precede provisioning by weeks or months. The hybrid model accommodates these constraints by separating infrastructure work into Kanban with longer planning horizons, but the integration of strategic infrastructure roadmaps with tactical Agile delivery remains an area where the practical tradeoffs are not well documented.

8.2.4 Scaling Agile Beyond 30 People

The case study site operates at 30 people, near the upper limit of what Agile (without Scaled Agile Framework or similar scaling overlays) can handle without significant additional structure. The Scaled Agile Framework, Disciplined Agile Delivery, Large-Scale Scrum, and Spotify-inspired tribe-and-squad models exist, but their validation in data engineering specifically is limited. Research on what happens to data engineering Agile patterns at the scale of 100 or more people would address a practical gap that organizations are encountering as their data functions grow.

8.3 Future Research Directions Several research directions emerge from the survey and case studies. First, more longitudinal case studies of Agile adoption in data engineering teams would build the evidentiary base that the field currently lacks. The two case studies presented here are instances; the value of more instances would compound. Second, standardization of metrics across organizations, including cycle time, throughput, and sprint goal completion rate, would enable cross-organization comparison that is currently impossible because different organizations measure differently. Third, the literature on Agile risk management is thin relative to its operational importance, and data engineering's high infrastructure risk makes the gap consequential; research on risk identification and mitigation in iterative delivery contexts would address a practical need. Fourth, platform engineering as a discipline is becoming increasingly important and increasingly visible, but the academic literature on platform team patterns, including the embedded squad model documented here, remains limited; case studies and pattern catalogs would help the field advance. Fifth, formal study of the organizational change management required to move from naive to adapted Agile in data engineering contexts, including the kinds of failure-and-recovery patterns that Sprint 7 illustrates, would provide guidance that current literature does not.

IX. CONCLUSION

==========

This paper has surveyed the Agile adoption patterns observed in data engineering teams, examined the challenges that drive adaptation away from software-derived patterns, and presented two longitudinal case studies that illustrate both the failures of naive adoption and the outcomes of considered adaptation. The contribution combines a structured account of the literature with practitioner evidence from a multi-year evolution at a financial services data organization.

The principal finding is that Agile works for data engineering, but only with intentional adaptation. The conditions that make software Agile workable (clear scope, testable outputs, deployment independence, low cross-team coupling) are only partially present in data engineering, and the patterns that succeed in data contexts are the patterns that acknowledge the differences rather than the patterns that pretend they do not exist. Pure Scrum is appropriate for small data engineering teams whose work resembles software work; hybrid Scrum-Kanban is appropriate for larger teams or teams with significant infrastructure dependencies; data mesh with platform enablement is appropriate at scale; the strangler fig pattern is appropriate for migration work regardless of the surrounding methodology. The choice among these patterns is principally a function of team size, infrastructure stability, and the organization's tolerance for parallel methodologies.

The case studies illustrate both the failure mode and the recovery. Sprint 7 in 2018 was the failure mode: a sprint that combined infrastructure migration with feature delivery, that was over-committed against unknown unknowns, that lacked formal risk identification and mid-sprint replanning, and that produced an 42 percent goal completion rate against a previously stable 65 percent baseline. The retrospective from that sprint produced the adaptations

(range estimation, risk backlogs, mid-sprint checkpoints, platform enablement squads, strangler fig migration, quarterly outcome commitments) that subsequently became the team's working model. The platform enablement squad case study from 2019--2020 illustrates the recovery and the cumulative effect of the adaptations: a 20 percent efficiency gain in domain team velocity, sprint goal completion rates in the 82 to 85 percent range, infrastructure latency for typical requests reduced from 2 to 3 weeks to 3 to 4 days, and cost-neutral first-year economics with positive ROI in subsequent years.

The case studies also surface lessons that the broader literature has not yet engaged with in depth. Sprint 7 illustrates that estimation refinement does not solve the underlying problem of commitment under uncertainty; the workable response is to commit less, identify risks more deliberately, and provide mechanisms for honest mid-sprint replanning when surprises emerge. The platform enablement squad case study illustrates that the team topology of an Agile organization matters as much as its methodology: embedding platform engineers with domain teams produces velocity gains that no methodology refinement could have produced on its own, because the underlying friction was structural rather than methodological. The organizational restructuring from eight direct reports to five, aligning team boundaries with the data mesh model of domain ownership plus platform enablement, was a precondition for the methodology adaptations rather than a consequence of them.

The cost-benefit picture is favorable in the case study context but should not be over-generalized. The 20 percent efficiency gain from the platform enablement squad reflects the specific baseline at the case study site and the specific causes of velocity loss in that environment; teams with different baselines and different causes should not expect the same magnitude of gain from the same intervention. The patterns described here are not a recipe; they are an existence proof that adapted Agile can work in data engineering contexts, accompanied by enough operational detail to make the patterns reproducible by teams whose contexts resemble the case study site's.

The closing observation, offered as a call to the practitioner community, is that data engineering Agile practices are not yet standardized in the way that software engineering Agile practices are. The literature is sparse, the metrics are inconsistent across organizations, and the failure modes are discussed less openly than they should be. Practitioners who have navigated similar transitions are encouraged to publish their experiences, including the failures, at the level of operational detail required to make the lessons transferable. Sprint 7 is a category of experience, not a singular event; the field will mature faster if more Sprint 7s are documented and analyzed in public rather than privately discussed and selectively forgotten.

REFERENCES

[1] D. J. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press, 2010.

[2] K. Beck, Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, et al., "Manifesto for agile software development," agilemanifesto.org, 2001.

[4] B. Boehm and R. Turner, Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, 2003.

[5] A. Cockburn, Agile Software Development. Addison-Wesley, 2002.

[6] M. Cohn, Agile Estimating and Planning. Prentice Hall, 2005.

[7] M. Cohn, Succeeding with Agile: Software Development Using Scrum. Addison-Wesley, 2009.

[8] K. Conboy, "Agility from first principles: Reconstructing the concept of agility in information systems development," Inf. Syst. Res., vol. 20, no. 3, pp. 329–354, 2009.

[9] Z. Dehghani, "How to move beyond a monolithic data lake to a distributed data mesh," martinfowler.com, 2019.

[10] K. Dikert, M. Paasivaara, and C. Lassenius, "Challenges and success factors for large-scale agile transformations: A systematic literature review," J. Syst. Softw., vol. 119, pp. 87–108, 2016.

[11] T. Dingsoyr, S. Nerur, V. Balijepally, and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," J. Syst. Softw., vol. 85, no. 6, pp. 1213–1221, 2012.

[12] T. Dyba and T. Dingsoyr, "Empirical studies of agile software development: A systematic review," Inf. Softw. Technol., vol. 50, nos. 9–10, pp. 833–859, 2008.

[13] M. Fowler, "StranglerFigApplication," martinfowler.com, 2004.

[14] M. Fowler and M. Foemmel, "Continuous integration," IEEE Softw., vol. 23, no. 4, pp. 75–78, 2006.

[15] M. Fowler and J. Highsmith, "The agile manifesto," Softw. Dev., vol. 9, no. 8, pp. 28–35, 2001.

[16] J. Grenning, "Planning poker or how to avoid analysis paralysis while release planning," Renaissance Softw. Consulting, 2002.

[17] R. Hoda, J. Noble, and S. Marshall, "Self-organizing roles on agile software development teams," IEEE Trans. Softw. Eng., vol. 39, no. 3, pp. 422–444, 2013.

[18] J. Highsmith, Agile Project Management: Creating Innovative Products, 2nd ed. Addison-Wesley, 2009.

[19] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley, 2010.

[20] G. Kim, K. Behr, and G. Spafford, The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win. IT Revolution Press, 2013.

[21] B. Kitchenham, "Procedures for performing systematic reviews," Keele Univ., Tech. Rep. TR/SE-0401, 2004.

[22] H. Kniberg and M. Skarin, Kanban and Scrum: Making the Most of Both. C4Media/InfoQ, 2010.

[23] C. Larman and V. R. Basili, "Iterative and incremental developments: A brief history," Comput., vol. 36, no. 6, pp. 47–56, 2003.

[24] C. Larman and B. Vodde, Large-Scale Scrum: More with LeSS. Addison-Wesley, 2016.

[25] D. Leffingwell, Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Addison-Wesley, 2010.

[26] V. Mahnic, "Improving software development through combination of Scrum and Kanban," in Proc. Recent Advances Comput. Eng., Commun. Inf. Technol., 2014, pp. 281–288.

[27] R. P. Maranzato, M. Neubert, and P. Herculano, "Moving back to Scrum and Scrumban," in Proc. 1st Int. Workshop Softw. Eng. Data Anal., 2012, pp. 17–20.

[28] S. McConnell, Software Estimation: Demystifying the Black Art. Microsoft Press, 2006.

[29] N. B. Moe, T. Dingsoyr, and T. Dyba, "A teamwork model for understanding an agile team: A case study of a Scrum project," Inf. Softw. Technol., vol. 52, no. 5, pp. 480–491, 2010.

[30] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," Commun. ACM, vol. 48, no. 5, pp. 72–78, 2005.

[31] M. Paasivaara, C. Lassenius, and V. T. Heikkila, "Inter-team coordination in large-scale globally distributed Scrum: Do Scrum-of-Scrums really work?" in Proc. 2012 ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas., 2012, pp. 235–238.

[32] K. Petersen and C. Wohlin, "A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case," J. Syst. Softw., vol. 82, no. 9, pp. 1479–1490, 2009.

[33] R. Pichler, Agile Product Management with Scrum: Creating Products that Customers Love. Addison-Wesley, 2010.

[34] M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit. Addison-Wesley, 2003.

[35] D. G. Reinertsen, The Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas Publishing, 2009.

[36] L. Rising and N. S. Janoff, "The Scrum software development process for small teams," IEEE Softw., vol. 17, no. 4, pp. 26–32, 2000.

[37] K. S. Rubin, Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley, 2012.

[38] K. Schwaber, "SCRUM development process," in Business Object Design and Implementation. Springer, 1995, pp. 117–134.

[39] K. Schwaber, Agile Project Management with Scrum. Microsoft Press, 2004.

[40] K. Schwaber and M. Beedle, Agile Software Development with Scrum. Prentice Hall, 2002.

[41] K. Schwaber and J. Sutherland, "The Scrum guide: The definitive guide to Scrum: The rules of the game," Scrum.org, 2017.

[42] M. Skelton and M. Pais, Team Topologies: Organizing Business and Technology Teams for Fast Flow. IT Revolution Press, 2019.

[43] I. Sommerville, Software Engineering, 9th ed. Addison-Wesley, 2010.

[44] C. J. Stettina and J. Horz, "Agile portfolio management: An empirical perspective on the practice in use," Int. J. Project Manag., vol. 33, no. 1, pp. 140–152, 2015.

[45] J. Sutherland, Scrum: The Art of Doing Twice the Work in Half the Time. Crown Business, 2014.

[46] H. Takeuchi and I. Nonaka, "The new new product development game," Harvard Bus. Rev., vol. 64, no. 1, pp. 137–146, 1986.

[47] CollabNet VersionOne, "13th annual state of agile report," 2019.

[48] K. Vlaanderen, S. Jansen, S. Brinkkemper, and E. Jaspers, "The agile requirements refinery: Applying Scrum principles to software product management," Inf. Softw. Technol., vol. 53, no. 1, pp. 58–70, 2011.

[49] J. Webster and R. T. Watson, "Analyzing the past to prepare for the future: Writing a literature review," MIS Quart., vol. 26, no. 2, pp. xiii–xxiii, 2002.

[50] L. Williams, "What agile teams think of agile principles," Commun. ACM, vol. 55, no. 4, pp. 71–76, 2012.

[51] M. Armbrust, et al., "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in Proc. CIDR 2021, 2021.