AGILE DEVELOPMENT OF OPTIMIZATION APPLICATIONS WITH AN EXPERIMENTAL BACKLOG

Peter Malacký[1]

Abstract: We describe how to apply agile and hypothesis-driven practices to optimization software development. By adding experimental backlog items that pair user stories with solver tasks, data tests, and clear hypotheses, each sprint delivers both a functional increment and a statistically validated improvement. The approach is grounded in agile values and evidence-based management.
Keywords: Agile development, optimization, hypothesis-driven development, backlog, data test

Introduction

Agile software development emphasizes iterative delivery, customer collaboration, and responsiveness to change [5]. In many industries, decision-support and optimization applications are critical for improving efficiency (e.g. logistics routing, scheduling, resource planning). Yet OR solutions often require extensive tuning and their real-world impact can be uncertain until measured. Traditional waterfall or ad hoc approaches may deliver a solver but not guarantee it yields better outcomes for users. We propose combining agile user stories with experimental methods so that every increment is also a small experiment testing a hypothesis about improvement. In our approach, each sprint does not just produce code, but also collects evidence that the change improves a key metric (e.g. cost, time, service level). This fusion of agile and hypothesis-driven development ensures validated learning: teams learn whether their solver changes actually deliver value before moving on [1][2].

The remainder of this article develops a structured methodology for this approach. We review agile and hypothesis-driven practices, discuss how to adapt them to optimization projects, and then describe practical processes and artifacts. We include templates and examples for backlog items (linking user stories to solver tasks and experiments), a sample sprint schedule, a flowchart of the process, and tables illustrating roles and criteria. Guidance is offered for different team sizes and solver types.

Background: Agile and Hypothesis-Driven Development

Agile development is grounded in the values of the Agile Manifesto: prioritizing individuals and interactions, working software, customer collaboration, and responding to change [5]. Agile teams break work into short iterations (sprints) and continuously incorporate feedback. A common framework is Scrum: a Product Owner maintains a product backlog of prioritized user stories, the development team implements features in time-boxed sprints (often 1–4 weeks), and each increment delivers potentially shippable software.

[1] Ing. Peter Malacký, Department of Communications, FPEDAS, UNiversity of Žilina, Slovakia, e-mail: peter.aalacky@uniza.sk

Traditional user stories follow the form "As a [role] I want [capability] so that [value]". However, in Hypothesis-Driven Development (HDD), each story is reframed as an experiment: teams explicitly state what they believe will happen and how they will measure it[4]. Barry O'Reilly (Lean Enterprise) suggests a template like:

We believe [this change/capability] will result in [expected outcome], and we will have confidence when [measurable signal of success][1].

For example: We believe that adding memory-caching to our route optimization engine will result in a 20% reduction in average solve time, and we will have confidence to proceed when we observe this speedup on sample workload tests.

In agile parlance, this means backlog items include not just feature descriptions but also hypotheses and metrics. After implementation, teams gather data (via A/B tests, benchmarks, user trials, etc.) to validate or refute the hypothesis [1][4]. Work is only "done" when the hypothesis has been tested and validated (or refuted) with real data. As O'Reilly notes, combining continuous delivery with hypothesis-driven development makes validated learning the primary measure of progress: "we should not say we are done until we have measured the value of what is being delivered"[1]. In short, agile plus experimentation ensures teams build the right solution, not just some solution.

Agile in Optimization and OR Projects

Applying agile methods to optimization/OR projects is gaining attention. Like data science, optimization development sits between research and engineering. Vidoni et al. (2020) surveyed agile adoption in OR, noting that both OR and software projects share similar phases (requirements, design, implementation, testing, deployment). They map agile artifacts (user stories, backlogs, releases) to OR artifacts (problem statements, mathematical models, solution outputs)[3]. For example, a Scrum-style product backlog of features can align with an OR backlog of mathematical models to build or refine, and acceptance tests can incorporate model validation or performance tests[3].

Crucially, Vidoni identifies team and project characteristics for agile in OR. Small teams (3–5 people) with limited agile experience may start with lightweight, incremental approaches (Type A methods), whereas larger or distributed teams (6–12+) can adopt fuller Scrum processes (Type B)[3]. This suggests tailoring the methodology: a small team might combine roles (the same person may handle both OR modeling and data analysis), while a medium team might have distinct roles for product owner, analyst, developer, and tester. Longer projects or those in rigid environments might integrate more documentation and formal planning, while short cycles focus on delivering incremental models and insights[3].

In all cases, the goal is to make optimization development iterative and empirical. Instead of delivering one big optimization model after long upfront work, teams break the problem into increments (for example, solve smaller subproblems or add features to the model in phases). After each sprint, they evaluate outcomes (e.g. solution quality, computational performance, business metrics). This mirrors agile's embrace of change and feedback, but with a twist: the feedback is quantitative validation of the solver's effectiveness.

Connecting User Stories, Solver Stories, and Experiments

User Stories and Solver Stories

In a standard Scrum project, a user story captures a customer need. For optimization projects, we similarly define user stories that reflect operational pain points. Example: "As a delivery dispatcher, I want the system to compute vehicle routes minimizing total distance, so that fuel costs are reduced." This story focuses on end-user value (cost reduction). The

development team then breaks it down technically. We introduce solver stories (or technical stories) as a way to document algorithm/model tasks derived from the user story. A solver story might be: "Implement a Mixed-Integer Programming (MIP) model for vehicle routing including time windows." This is not phrased as an end-user need but as a technical task aligned with the user story's goal.

By linking these, we ensure traceability: each solver story has a parent user story. For example, the solver story above explicitly supports the dispatcher's need. In agile terms, the product backlog contains both user stories (customer-facing features) and technical backlog items (solver/model tasks). Each item should have acceptance criteria. For a user story, acceptance might involve both functional verification (the UI shows routes) and metric verification (routes are within X% of optimal). For a solver story, acceptance would include correctness (the model solves test instances) and performance (solution time or quality meets expectations).

The key is that solver stories incorporate algorithmic improvements: choosing heuristics vs exact methods, tuning parameters, adding constraints, etc. Example solver story: "As a developer, I want to implement a novel heuristic that uses historical traffic data to seed initial routes". Its acceptance criteria might include benchmark results on historical datasets. In effect, the development team views algorithm development as a sequence of agile stories, with each increment building and testing some aspect of the solver.

This approach contrasts with traditional OR development where a model might be fully built and only then tested. Instead, each sprint delivers a partial solver enhancement that is immediately tested.

Hypotheses and Data Tests

Every backlog item (user or solver story) carries an explicit hypothesis about improvement. Following the hypothesis-driven template[4], we require something like: "We believe that [change] will result in [quantified outcome], and we will have confidence when [metric threshold] is met." For the dispatcher example:
We believe adding historical traffic data to the routing solver will reduce average delivery time by 15%. We will have confidence in this hypothesis when, in test simulation on last month's orders, the average time reduces by at least 15% compared to the baseline solver.

The hypothesis frames the experiment. Next, a data test plan specifies how to measure: this might be an A/B test in production, a simulation using historical data, or a dedicated benchmark. For instance, one could "run the legacy solver and the new solver on a 10-day sample of routes and compare metrics." The data test identifies datasets, metrics, and analysis method. Common metrics include percentage improvement in cost, time, or service level, and statistical significance criteria for A/B tests. (A data scientist or QA engineer typically designs these tests.) For optimization tasks, data tests often simulate real scenarios: e.g. backtesting, Monte Carlo, or pilot deployment.

By codifying these, each sprint yields not only "working code" but also empirical results. If the hypothesis is confirmed (success metric reached), the feature is validated. If not, the team learns and can pivot: perhaps the approach needs revision or a different assumption. Importantly, this ensures we don't continue down unpromising paths indefinitely – unvalidated assumptions surface early.

Evidence-based management frameworks also suggest measuring outcomes (e.g. customer satisfaction, business value) rather than outputs[2][1]. In our method, that means every acceptance criterion includes not just technical correctness but evidence of value. For example, "Total delivery time reduced by ≥15% in pilot" rather than just "routes computed within 5s".

Experimental Backlog Methodology

We now outline how a sprint-based process incorporates these ideas. The heart of the methodology is the experimental backlog: a product backlog that mixes regular features with items framed as experiments (hypotheses). As Scrum.org notes, we should "include 'experimental' backlog items…work driven by a clear future-looking hypothesis, work that has clearly defined value"[2]. In practice:

1. **Backlog Refinement**: The Product Owner (often with input from data analysts and stakeholders) writes or refines backlog items. Each item has: (a) a description (user or solver story), (b) an explicit hypothesis ("We believe X yields Y"), (c) a data test plan, and (d) a quantifiable success metric. Tools like user story templates can be extended to include these fields.
2. **Sprint Planning**: The team selects a set of backlog items for the sprint, balancing feature development, algorithm improvements, bug fixes, and experiments. For each experimental item, plans are made to gather data (e.g. reserve test cases, schedule an A/B run).
3. **Implementation**: Developers and optimization experts implement the feature/solver enhancements. Concurrently, data engineers prepare data sets and test infrastructure needed to evaluate them.
4. **Execution and Testing**: Once implemented, the team immediately runs the defined data tests. This could be automated benchmarks, simulation runs, or A/B testing in a controlled environment.
5. **Review**: The sprint review includes not only a demo of new functionality but also a presentation of results: e.g. performance graphs, statistical analysis, or pilot outcomes. The hypothesis is either accepted (if metrics are met) or investigated further.
6. **Retrospective & Learning**: The team reflects on both the development process and the experimental outcomes. If a hypothesis failed, the next sprint's backlog may include refined experiments or new ideas.

Key to this workflow is that "Done" means tested for value. For example, Definition of Done for a solver story might be: code is merged, unit tests pass, and experiment confirms the expected improvement (or clearly documents the result). This ensures validated learning becomes part of the Definition of Done[1].

Roles, Artifacts, and Acceptance

The experimental backlog approach involves both traditional agile roles and roles specific to OR:

Table 1. Roles, artifacts, and example acceptance criteria in the experimental backlog methodology.

Role	Artifacts/Deliverables	Acceptance Criteria
Product Owner / Stakeholder	Product Backlog (with prioritized user stories and experiments), product vision	Stories with clear value hypotheses; backlog reflects stakeholder goals
Scrum Master / Coach	Sprint backlog, burndown charts, retrospective notes	Team follows process; impediments removed; process improvements made
Developer / OR Modeler	Working code (software components, solver modules), mathematical models	Code compiles; unit tests and model validations pass; satisfies solver constraints

Role	Artifacts/Deliverables	Acceptance Criteria
Data Scientist / Analyst	Test datasets (real/synthetic), benchmark results, A/B test reports	Data tests are executed; metrics collected; statistical analysis done
Tester / QA	Acceptance test scripts, regression tests	All functional tests pass; new features meet performance targets

Defining Done: Functionality and Validated Improvement

Traditionally, "Done" for a user story means the feature is coded, tested, and deployed. Here we add a scientific element: observing the intended outcome. A useful guideline (inspired by Evidence-Based Management) is that progress is measured in customer/business outcomes[2]. Thus, after a sprint, the team should be able to point to data confirming improvement. If not, they have learned that more work is needed or the assumption was wrong.

Consider an example acceptance for a backlog item: "Route computation time must be under 10 seconds and average delivery distance must decrease by at least 5% on our test set." Both parts are mandatory. If the code meets the first criterion but not the second, the item is not done in our methodology – the hypothesis is unproven. This may generate new backlog items (e.g. "investigate why distance did not drop").

This tight linkage between building and measuring increases accountability and aligns the team with business goals. It also justifies the sprint investment: each sprint yields both software and insight (validated or not), rather than producing features of unknown value.

Example Backlog Table

Table 2 shows sample backlog items combining user stories, solver stories, and data tests. Each row has an ID, type, description, hypothesis, data test, success metric, priority, and planned sprint. This illustrates how to record and prioritize experimental items.

Table 2. Sample backlog items (US=user story, SS=solver story, DT=data test) with linked hypotheses and metrics.

ID	Type	Description	Hypothesis	Data Test	Success Metric	Priority	Sprint
US1	User Story	As a dispatcher, I want the system to optimize delivery routes to minimize distance.	Using advanced routing will reduce avg. distance by ≥15%.	Compare distances on historical orders (old vs new solver).	≥15% reduction in avg. distance	High	1
SS1	Solver Story	Enhance routing solver: integrate real-time traffic data as	Including traffic data will cut travel time by ≥10%.	Simulate routes with/without traffic data on sample cases.	≥10% drop in avg. travel time	High	1

ID	Type	Description	Hypothesis	Data Test	Success Metric	Priority	Sprint
		additional cost factor.					
DT1	Data Test	Benchmark current solver on baseline dataset of 100 cases.	*N/A (baseline measurement for comparison)*	Run current solver on 100 cases; record metrics.	Record baseline distance, time, cost	Medium	1
US2	User Story	As an operator, I want scheduling to consider crew availability so that schedules are feasible.	Constraint programming solver will ensure 100% feasible schedules.	Test new CP model on 200 crew scenarios.	100% schedules feasible (no violations)	Medium	2
SS2	Solver Story	Implement CP-SAT model for scheduling with crew constraints.	CP-SAT yields solutions 25% faster than current heuristic on average.	Run CP-SAT vs old solver on 50 sample tasks.	≥25% faster average solve time	High	2

Each item in the backlog table spans multiple categories: types (user, solver, experiment) and fields ensure traceability. Prioritization depends on business impact and feasibility. For instance, US1 and SS1 are high priority to quickly validate the core routing improvement. US2 and SS2 may be scheduled in Sprint 2 after initial successes.

Guidance for Team Sizes and Solver Types

Team Size. Small teams (3–5 people) should keep processes lightweight. Roles may be combined: for example, one person might be both product owner and analyst, or a developer might write and run experiments. Sprints might be 1 week long to maintain focus. Communication is easy in co-located small teams, but they should still explicitly allocate time to run experiments (not all developers may be experienced with statistical testing). Large or medium teams (6–12) can separate roles: a dedicated analyst can design experiments, and a scrum master can facilitate. Sprints of 2–3 weeks allow for more complex work. Medium teams should also maintain a living backlog document with hypotheses and metric results clearly recorded (for knowledge sharing). In either case, cross-functional training (developers learning basic stats, analysts understanding code) helps.

Solver Type. The development process can adapt to different solver approaches:

- **Heuristic/Metaheuristic:** These solvers often evolve incrementally (e.g. adding a new local search operator). The backlog can capture each heuristic enhancement as a story, with experiments comparing solution quality or speed. A/B tests might be Monte Carlo

simulations. Because heuristics may have randomness, multiple runs and statistical measures (mean, variance) should be used.

- **Mixed-Integer Programming (MIP):** Building an MIP model (e.g. in Gurobi) is often heavy but can be modularized. Early sprints might implement a basic model without all constraints (validating feasibility), then iteratively add constraints (time windows, capacities) in later sprints. Each addition is a solver story with a hypothesis (e.g. "adding constraint X will increase solution time by ≤ Y%"). Data tests involve solving benchmark instances; success metrics cover optimality gaps or solve times.
- **Constraint Programming (CP):** CP models (e.g. OR-Tools CP-SAT) similarly are built iteratively. CP is often used when feasibility is tricky; initial stories might focus on correct constraint encoding. The incremental approach works well here, too: each sprint adds or refines a constraint or heuristic in the CP search. Metrics involve constraint violation rates or solve speed.

The common thread is that no matter the technique, backlog items describe both what is built and how its effect will be evaluated. For instance, a solver story for a heuristic might hypothesize a quality improvement, whereas a MIP story might focus on performance or robustness.

Conclusion

Integrating agile development with an experimental mindset is essential for building optimization applications that deliver real value. By structuring backlog items as hypothesis-driven experiments, teams make data-driven decisions at every sprint. This approach aligns with agile values of collaboration and responsiveness[5] while adding a scientific rigor to validation[1][2]. The methodology presented here – roles and artifacts tables, backlog templates, sample sprint plan, and diagrams – provides a practical guide for teams of varying size.

In practice, this means no more “just fixing bugs or adding features” in optimization projects; every change is asked, “How will we know this improved things?”. The examples and templates can be adapted: for instance, an airline crew-scheduling team might use CP stories and test with historical rosters; a supply-chain planner might add inventory heuristics and test with rolling forecasts.

Ultimately, every sprint concludes not just with working software but with validated learning: evidence that the optimization solution moves the needle on key metrics. Over successive sprints, this leads to continual operational improvement and high confidence in the optimization software’s impact.

References

[1] O’REILLY, B.: How to Implement Hypothesis-Driven Development, Barry O’Reilly, 2013, dostupné online: https://barryoreilly.com/explore/blog/how-to-implement-hypothesis-driven-development/, cit. 16. 6. 2026.

[2] SCRUM.ORG and DEVOPS INSTITUTE: The Convergence of Scrum and DevOps, Scrum.org, 2017, dostupné online: https://www.scrum.org/resources/convergence-scrum-and-devops, cit. 16. 6. 2026.

[3] VIDONI, M., CUNICO, M. L., VECCHIETTI, A.: Agile operational research, Journal of the Operational Research Society, 2021, roč. 72, č. 6, s. 1221–1235, DOI: 10.1080/01605682.2020.1718557.

[4] YAN, E.: Data Science and Agile: What Works, and What Doesn’t, 2019, dostupné online: https://eugeneyan.com/writing/data-science-and-agile-what-works-and-what-doesnt/, cit. 16. 6. 2026.

[5] KOCÁKOVÁ, Z.: Čo je agilný vývoj a aké sú agilné metodiky riadenia softvéru?, 2022, dostupné online: https://msgtester.sk/agilny-pristup/, cit. 16. 6. 2026.
[6] SCHWABER, K., SUTHERLAND, J.: Sprievodca Scrumom: Definitívny sprievodca Scrumom – pravidlá hry, Scrum Guides, november 2020, dostupné online: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Slovak.pdf, cit. 16. 6. 2026.
[7] DYBÅ, T., DINGSØYR, T.: Empirical studies of agile software development: A systematic review, Information and Software Technology, 2008, roč. 50, č. 9–10, s. 833–859, DOI: 10.1016/j.infsof.2008.01.006.
[8] MINGERS, J., ROSENHEAD, J.: Problem structuring methods in action, European Journal of Operational Research, 2004, roč. 152, č. 3, s. 530–554, DOI: 10.1016/S0377-2217(03)00056-0.
[9] KOHAVI, R., LONGBOTHAM, R., SOMMERFIELD, D., HENNE, R. M.: Controlled experiments on the web: Survey and practical guide, Data Mining and Knowledge Discovery, 2009, roč. 18, č. 1, s. 140–181, DOI: 10.1007/s10618-008-0114-1.
[10] LIU, Y., BOSCH, J., OLSSON, H. H., and LANTZ, J.: An architecture for enabling A/B experiments in automotive embedded software, In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid: IEEE, 2021, s. 992–997, ISBN 978-1-6654-2463-9, DOI: 10.1109/COMPSAC51774.2021.00134.
[11] BEHUTIYE, W., KARHAPÄÄ, P., LÓPEZ, L., BURGUÉS, X., MARTÍNEZ-FERNÁNDEZ, S., VOLLMER, A. M., RODRÍGUEZ, P., FRANCH, X., and OIVO, M.: Management of quality requirements in agile and rapid software development: A systematic mapping study, Information and Software Technology, 2020, roč. 123, čl. 106225, DOI: 10.1016/j.infsof.2019.106225.