Whitepaper

Learning cycles

How to predict and accelerate advanced technology development

Innovation series.

A practical method for predicting and accelerating breakthroughs in advanced technology development. Built around the T.J. Rodgers easy/hard/difficult sizing model and the lateralworks Fast Time To Market (FTTM) methodology.

Prepared by	Date	Online
lateralworks	May 2026	lateralworks.com
FTTM methodology	Methodology paper	Innovation series

Table of contents

Learning cycles

Core thesis. Innovation is not random. The duration of a breakthrough can be approximated by counting learning cycles and multiplying by typical cycle time, sized against problem difficulty. Programs accelerate by working two levers in parallel: *more cycles running concurrently* and *shorter cycle time per cycle*. The first lever doubles or quadruples the rate of learning. The second compresses each unit of learning. Together they convert "we cannot predict when" into "we predict it within this band, and here is how we will compress it."

Overview

Abstract

The objection to planning innovation has a clear answer. Teams pushing the frontier of physics, chemistry, or software architecture often refuse to commit to schedules on the grounds that no one can predict when a breakthrough will arrive. The argument has surface logic. It has also cost programs years of avoidable delay [1, 2]. Innovation is a sequence of learning cycles whose count and duration can be estimated and compressed with the same discipline that governs ordinary engineering work.

This paper presents the lateralworks method for predicting and accelerating breakthroughs in advanced technology development. The method draws on a sizing technique originated by T.J. Rodgers at Cypress Semiconductor and refined across more than 200 engagements at lateralworks since 1988 [3]. Every problem is classified as easy, hard, or difficult. Each class carries a typical number of learning cycles and a typical duration per cycle. The product of the two yields a rough but honest estimate of total breakthrough time.

The estimate is the start, not the end. Acceleration follows from two levers. The first is concurrency. Most programs run learning cycles serially when they could run four, six, or eight in parallel; the parallel arrangement converts wall-clock time into multiplied learning capacity [4, 5]. The second is cycle time. Faster cycles shorten the unit of learning, which compounds across the count. Set-based concurrent engineering at Toyota, Reinertsen's work on flow, and Thomke's work on rapid experimentation all converge on the same conclusion: in uncertain domains, more iterations of smaller units beat fewer iterations of larger ones [6, 7, 8].

The remainder of the paper walks through nine working components. Sections 01–03 establish the framework: why innovation can be planned, how learning cycles are defined, and how the easy/hard/difficult sizing produces useful estimates. Section 04 frames the two acceleration levers. Sections 05–08 cover the supporting practices — fast-team behavior, refresh planning, isolation of the breakthrough work, and the use of learning milestones rather than feature milestones for advanced programs. Section 09 collects the method into a nine-step implementation playbook a team can adopt within a single quarter.

The point of the paper is operational, not theoretical. Programs that adopt this method routinely deliver advanced technology in 60–70 percent of the time their original schedules forecast, with predictions accurate enough to inform capital allocation, market timing, and competitive response. The barrier is not technical. It is the cultural reflex that says "we cannot know yet" and uses that claim to defer the planning work that would close the gap.

01

The myth

Innovation can be planned

The most common objection to planning advanced technology development is that innovation is fundamentally unpredictable. Teams say it. Senior leaders say it. Even investors say it when products miss their target market windows. The argument sounds reasonable. It is also wrong. Innovation is unpredictable in the small — no one knows exactly which design of experiment will yield a breakthrough on a given Tuesday. It is far more predictable in the aggregate, and that is the level at which schedules need to operate.

Before exploring how to plan it, it helps to understand how innovation arrived at this position in the first place. Research and development used to be two distinct activities with their own time horizons and their own management practices. They are no longer separated, and the schedule pressure that followed is the root cause of most of the "innovation cannot be planned" pushback teams produce today.

Section 01 — continued

How research and development merged

Through the 1980s, research happened in laboratories and was not scheduled. It produced findings when it produced them. Development happened in product teams with budgets and target launch dates. The two activities were sequential. Research ran first, often for five to seven years; development followed, with a separate three to five-year horizon [9]. The arrangement worked because customers and competitors tolerated long cycles, and because development teams inherited a reasonably mature technology to package and ship.

That window closed. By the late 1990s, average R&D cycle time in Silicon Valley had compressed from five-to-seven years to ten-to-fifteen months. In storage systems, it dropped below six months. In some software environments it fell further [10]. The pressure came from competitive cycles, customer expectations, and the rising capital cost of long research programs. Research departments shrank or disappeared. Research became a thing that happened inside development projects, on the development team's schedule, with the development team's budget. The activities merged, but the management practices did not.

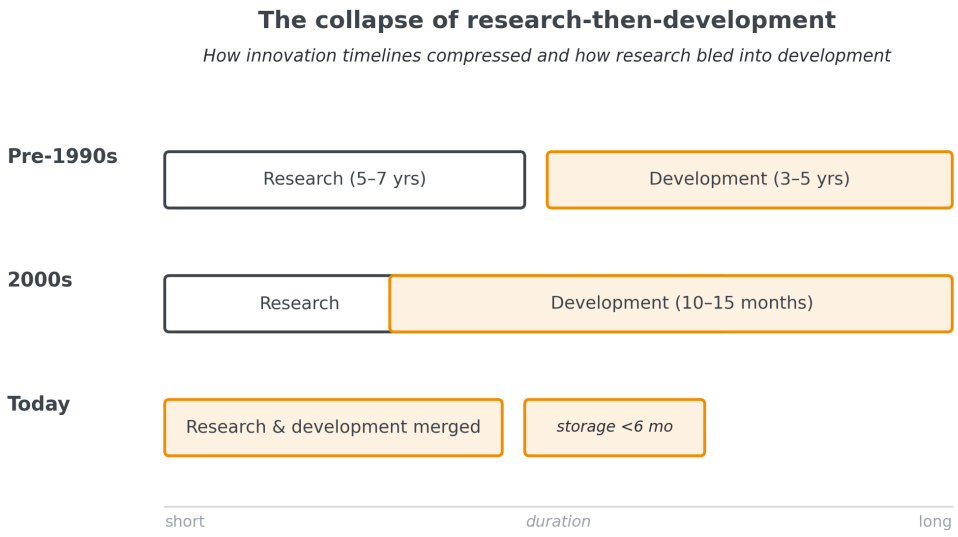

Figure 1. Research and development used to be sequential and long. They are now merged and short. Teams inherited the schedule pressure of the second without the time horizon of the first.

The consequence is visible in nearly every program lateralworks audits. A team is asked to commit to a launch date measured in months. Inside that envelope, three or four genuine breakthroughs are required — a new manufacturing process, a new device, a new software architecture, a new firmware approach. The team pushes back, correctly, that the schedule does not contain the research time those breakthroughs need. Management replies, equally correctly, that the market window does not accommodate a longer schedule. Both are right. Neither side has a method for closing the gap.

This stalemate is what learning cycles resolve. The method does not promise that breakthroughs will arrive faster than physics permits. It does promise an honest estimate of how long the breakthrough work will take, a structure for compressing that estimate, and a way to surface the schedule gap early enough to act on it — by narrowing scope, isolating the innovation work, or adjusting market commitments. Edmondson reaches a parallel conclusion in her work on learning organizations: teams that surface uncertainty early outperform teams that mask it [11].

Why "we cannot predict" became the default answer

Three things drive the reflex. The first is honest uncertainty. No one knows which experiment will succeed. The second is incentive structure. Teams that commit to a date and miss it are punished; teams that decline to commit cannot miss. The third is the absence of a working method. Without a way to size the unknown, "we cannot predict" is the only answer that survives scrutiny [12]. The learning-cycle method addresses the third reason directly, and in doing so changes the calculation on the second.

02

Mechanism

How learning cycles work

A learning cycle is the smallest unit of progress in advanced technology development. Plan an experiment, run it, check the result, act on what was learned. The shape is borrowed from Shewhart's 1939 work on statistical process control and from Deming's extension of it in postwar Japan [13, 14]. The application to breakthrough innovation, with explicit sizing of cycle count and cycle duration, was developed by T.J. Rodgers at Cypress Semiconductor and transmitted to lateralworks via Tony Alvarez, who ran R&D at Cypress for many years [3].

The method is simple enough to fit on a single sheet of paper and rigorous enough to schedule a 14-nanometer semiconductor process or a quantum-communication product. What follows is the working anatomy.

Section 02 — continued

The anatomy of a learning cycle

Every learning cycle has four stages. Plan defines the question the cycle will answer and the experiment that will answer it. Do executes the experiment. Check inspects the result against the hypothesis. Act decides what to do with what was learned: refine the next cycle, eliminate a branch, or close the question. The full loop is the unit of currency in advanced technology development.

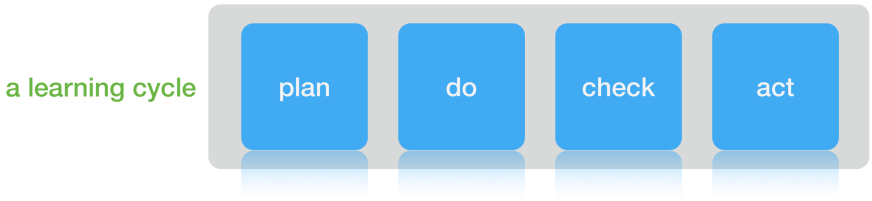

Figure 2. A single learning cycle. Plan the experiment, run it, check the result, act on what was learned. The cycle is the basic unit of progress for any problem whose solution is unknown at the start.

The discipline of the four stages matters more than the four words. A team that runs an experiment without an explicit hypothesis cannot interpret the result. A team that interprets a result without committing to an action wastes the learning. A team that acts without revisiting the plan repeats the work. Each stage exists because the others depend on it.

Learning means failing — at speed

Failure inside a learning cycle is the output, not a setback. A cycle that confirms a working hypothesis closes one question and reveals the next; a cycle that disproves a hypothesis closes the same question with the opposite answer. Both are useful. The cycle that produces no clear signal wastes time, and the design discipline above is what prevents it.

This is the basis for the phrase "fail fast." It is misused widely. The phrase does not mean shipping sloppy or broken work. It means compressing the time between hypothesis and answer so the team can move to the next question. Thomke documents this dynamic across pharmaceutical, semiconductor, and software firms: the firms that learn fastest are those that run the largest number of cleanly designed experiments per unit time, not those that run the cleverest single experiment [7].

Origin note. The easy/hard/difficult sizing model that follows in Section 03 was originated by T.J. Rodgers at Cypress Semiconductor and taught to development teams there in the 1990s. Tony Alvarez, who managed Cypress R&D, introduced the method to lateralworks. It has been refined across more than 200 engagements since [3]. The Plan-Do-Check-Act loop itself dates to Walter Shewhart in 1939 and was popularized by W. Edwards Deming after World War II [13, 14].

03

Estimation

The easy, hard, difficult method

The estimating method has two inputs and one output. Difficulty is classified as easy, hard, or difficult. Cycle duration is estimated from typical experiment time for the domain. The output is the expected number of cycles needed to find a solution, multiplied by the cycle duration, expressed as a range rather than a single number. Precision is not the goal. Honesty is.

Experienced engineers, asked to classify a problem they have not yet solved, are surprisingly accurate. The work of classification itself surfaces what the team knows and does not know, which is the first step in compressing the schedule.

Section 03 — continued

Three difficulty buckets

The three buckets are deliberately coarse. They do not need to be more granular to be useful. Easy problems are those for which a solution path is visible at the start — the technology has been demonstrated elsewhere, or the team has solved analogous problems before. Hard problems require real experimentation: multiple cycles, plausible failure paths, no single obvious solution. Difficult problems are at the frontier — the answer is not known to exist, or has only been suggested theoretically.

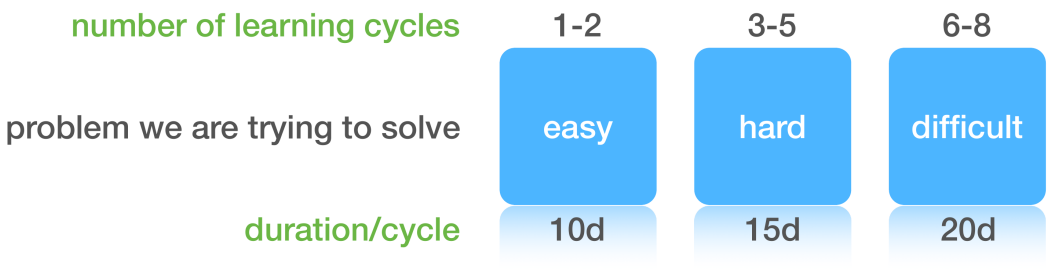

Figure 3. The easy, hard, difficult sizing model. Each class carries a typical number of cycles and a typical duration per cycle. Use your own numbers calibrated to your domain; the structure is what matters.

The numbers shown in the figure are the lateralworks defaults, calibrated against semiconductor, storage systems, and software programs. Other domains will use other numbers. A wet chemistry program may have cycles measured in weeks; a firmware program may run cycles in days. Calibrate against the team's own recent project history rather than borrowing other teams' numbers. The method works regardless of the specific values — what matters is that each class has consistent values across the program.

A worked example

A team is asked to estimate a device-level breakthrough. After discussion, the team classifies the problem as hard, on the easier end of hard, leading to an estimate of three learning cycles. Cycle time for this domain is fifteen days. The estimate is three times fifteen, or forty-five calendar days. The team commits to this estimate with the explicit acknowledgment that the actual count may be two cycles or four; the variance comes out in tracking, not in the estimate itself.

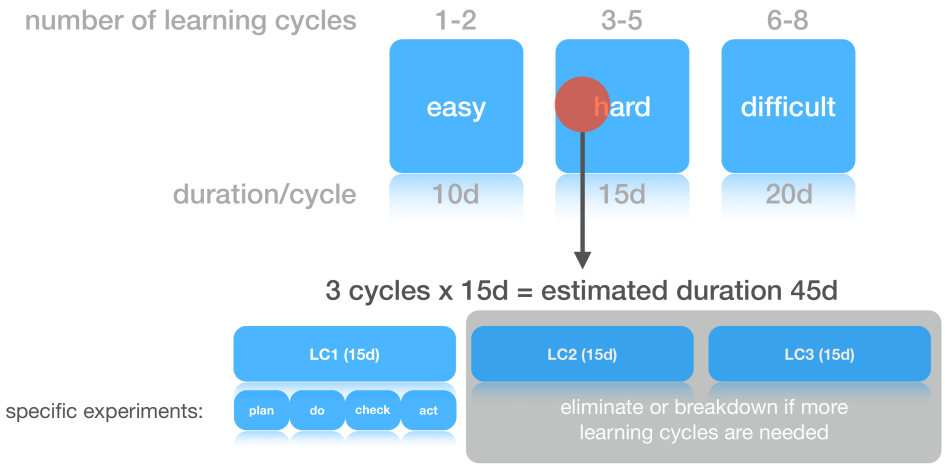

Figure 4. A worked estimate. Three cycles of fifteen days each yields a forty-five-day estimate. Each cycle is planned in detail only when the team reaches it; the placeholder cycles can be eliminated or broken down further depending on what the first cycle reveals.

Two observations carry over from the example. First, the team only details the next cycle. Cycles two and three are summary placeholders until cycle one completes. Detailing future cycles before knowing what cycle one will produce is a common waste, and one that produces false precision. Second, the estimate is committed to as a band, not a point. "Forty-five days, plus or minus fifteen" is honest. "Forty-five days exactly" is not.

> **Calibration tip.** Run the easy/hard/difficult assignment on three or four recently completed advanced-development efforts before applying it to active programs. Compare the model's prediction to actual cycle counts and durations. Adjust the bucket numbers until the model retro-predicts the past within ±20 percent. The model is then ready to use forward.

Reflection

On planning what you do not yet know

Roughly right beats exactly wrong.

lateralworks engagement principle.

Quantified guesses beat unquantified hope.

04

Acceleration

The two levers

A schedule estimate is only the start. Once a program knows how long the breakthrough work would take if run in the conventional way, the next question is whether that estimate fits the market window. It almost never does. The gap is the working space, and the method for closing it is what separates the programs that ship from the programs that miss.

There are exactly two levers. Increase the number of concurrent cycles. Reduce the duration of each cycle. Most programs use neither. Programs that close the gap reliably use both, with the first applied first.

Section 04 — continued

More cycles, shorter cycles

The first lever is concurrency. A program with a single learning path delivers one cycle of progress per cycle duration. A program with four concurrent paths delivers four. The arithmetic is obvious; the practical consequence is not. Concurrency requires the program to invest in dedicated resources, parallel experimentation infrastructure, and disciplined integration of results. Reinertsen frames this as a queueing problem: limit work in progress at the individual cycle level, but increase total throughput by parallelizing across paths [6]. Iansiti and MacCormack documented the same dynamic in their study of Internet-era software development — the firms that shipped fastest were the firms that pursued multiple architectural options in parallel and converged late, not the firms that picked a single path early [4, 5].

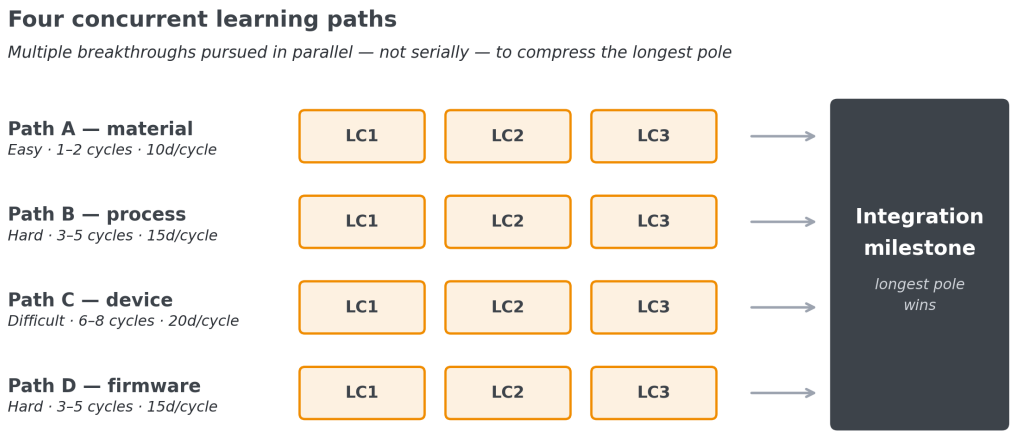

Figure 5. Four concurrent learning paths. The longest pole wins, but having four poles compresses the expected longest pole more than having one.

Set-based concurrent engineering at Toyota carried this principle into automotive design. Ward, Liker, Cristiano, and Sobek named it the "second Toyota paradox": delaying decisions can make better cars faster, because each delayed decision is made with the benefit of evidence from parallel exploration rather than a guess made on day one [15, 16]. The semiconductor industry has converged on a similar pattern under different names — typically called design space exploration, parallel DOE, or split-lot experimentation. The vocabulary differs; the structure is the same.

The cycle-time lever

The second lever is cycle duration. A team that compresses each cycle from twenty days to ten days doubles its learning rate per path. Combined with four-fold concurrency, the result is eight times the conventional learning capacity per unit of wall-clock time. Compression has its own cost — it usually

requires investment in faster prototyping, more characterization equipment, more compute, or more skilled people — but the cost-of-delay calculation almost always justifies it [6, 17].

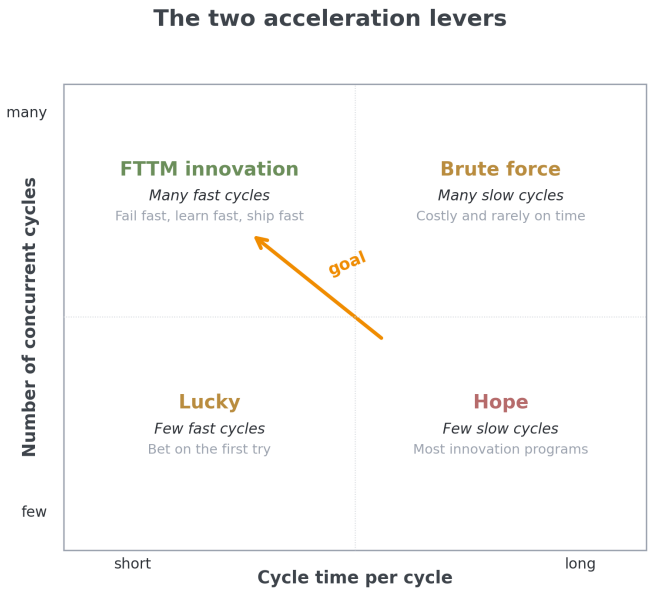

Figure 6. The two levers as a quadrant. Most programs sit in the bottom-right: few slow cycles, hope. The target is the top-left: many fast cycles, with the orange arrow showing the direction of progress.

The quadrant makes one further point. The path from bottom-right to top-left does not go through the corners. A program that adds concurrent paths without compressing cycles moves to the brute-force quadrant: costly, slow, and rarely on time. A program that compresses cycles without adding paths moves to the lucky quadrant: it learns faster on the single path it chose, but a wrong path choice on day one still dominates the outcome. The two levers reinforce each other only when both are pulled together.

05

Behavior

Slow teams, fast teams

Compressing cycle time is a team-behavior problem before it is a tooling problem. Two teams with identical equipment, identical staffing, and identical problem definitions can produce wildly different cycle counts in the same wall-clock window. Talent is rarely the difference. The difference is how the team is structured to handle failure.

Slow teams optimize for "right the first time." Fast teams optimize for "wrong as fast as possible, then right." The choice produces an order-of-magnitude difference in learning rate.

Section 05 — continued

Do it, try it, fix it

The lateralworks shorthand for fast-team behavior is "do it, try it, fix it." The team builds the simplest viable version of the experiment, runs it, learns what it can, and iterates. Polishing the experiment before running it adds delay and rarely adds insight. The first run almost always reveals something the team did not anticipate, which is the entire point of running it.

Figure 7. Two teams running in the same wall-clock window. The slow team completes one long cycle; the fast team completes three. The slow team's single cycle was not three times as informative.

Edmondson's research on team learning explains the underlying mechanism. Teams that learn fastest operate with high psychological safety — a shared belief that interpersonal risk is acceptable. Failure is reportable. Failure is discussable. The team is judged on whether it learns from each cycle, not on whether each cycle succeeds [11, 18]. Teams without that environment hide partial failure, repeat work to mask it, or wait until a cycle is "ready" before submitting it. All three patterns slow learning by a factor of two or more.

Why the slow-team pattern persists

Slow-team behavior is rational inside a punishing corporate culture. If a failed cycle generates a performance review issue, the team learns to ship only cycles likely to succeed. Cycles likely to succeed are the cycles least likely to produce learning. The system selects itself into a state where the team is busy, the dashboards are green, and the program is months behind. Reinertsen observes the same phenomenon at the project portfolio level: the metrics that feel safest to track are the ones that destroy economic value [6].

Changing this starts with leadership behavior, not process. Senior leaders model the kind of failure they want to see by reporting their own learning publicly, by asking "what did this cycle teach us?" before "why did it fail?", and by reviewing trend lines across cycles rather than pass/fail status on the most recent one. lateralworks engagement data shows that programs which adopt these three leadership behaviors compress cycle time by 30–50 percent within a single quarter, with no change in staffing or tooling [3].

Practical test. Ask the team: "When was the last time a cycle produced a result you did not expect?" If the answer is "we cannot remember," the team is running confirmatory work rather than learning cycles. Compression is not possible until that changes.

06

Planning

Refresh and incremental

A schedule that estimates breakthrough time is only useful if it tracks reality. A schedule that does not update as the team learns is wrong by the second week. The lateralworks remedy is a weekly refresh discipline: detail what is known now, leave the future as macro placeholders, and roll the detailed window forward each week as the next set of work becomes clear.

This is the opposite of the build-3500-tasks-on-day-one approach that traditional project management produces. It looks less precise. It is more accurate, because it is honest about what the team does and does not know.

Section 06 — continued

The rolling window

The pattern is straightforward. The near term — typically the next four to six weeks — is detailed to micro tasks of five days or less. The mid term is held as macro chunks the size of a learning cycle or a milestone. The far horizon is a single placeholder, the macro plan to product launch. Every week, the team rolls the window forward: tasks that completed drop off, the mid-term chunk that is now imminent breaks down into micro detail, and a new macro chunk appears at the far end. The total plan size stays roughly constant; the location of detail moves.

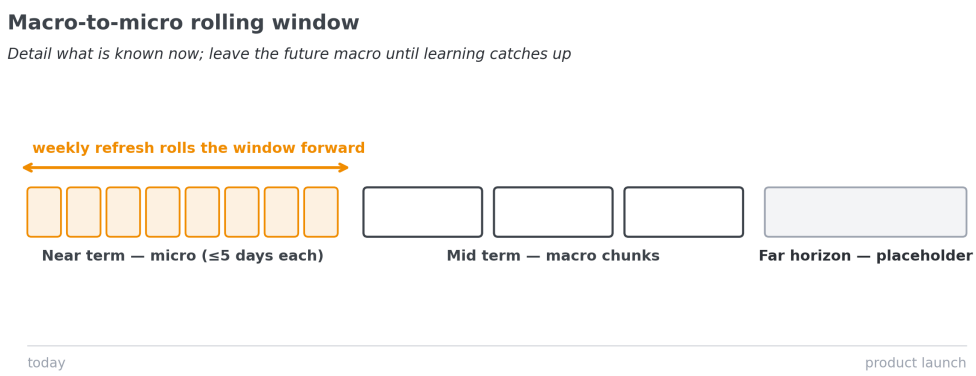

Figure 8. The rolling window. Detail concentrates in the near term and rolls forward weekly. The far horizon remains macro until learning catches up.

The objection is reliable. "If we do not detail the back end, how do we know when we will ship?" The answer is that the macro chunks carry duration estimates sized using the easy/hard/difficult method. The macro plan integrates to a launch date with a confidence band. The band tightens as more of the future moves into the detailed near term and the easy/hard/difficult estimates are replaced by observed cycle times. Detail in the far horizon is false precision; it does not tighten the confidence band, and it generates rework every time the architecture changes.

Refresh cadence — weekly, not monthly

The refresh is weekly because most learning cycles produce information on a weekly or sub-weekly basis. A monthly refresh accumulates four weeks of stale plan, which is enough drift to make the plan misleading. A weekly refresh keeps plan and reality within one cycle of each other. Anderson’s work on Kanban and Reinertsen’s work on cadence both converge on the same conclusion: the planning cadence should match the rate at which new information arrives [6, 19].

The refresh is also short. lateralworks-run refreshes are typically 60 to 90 minutes for a complex semiconductor program. The agenda is fixed: what completed, what is starting, what changed in the macro estimates, and what does the new critical path look like. Status review is not part of the meeting; it is

asynchronous, pre-read, and visible on a shared dashboard. The meeting is for replanning, not for reporting.

07

Concentration

Isolate and focus

A program described as "an innovation project" is rarely all innovation. In lateralworks engagement data across more than 200 advanced-development efforts, the average breakthrough work is roughly ten percent of the total scope; the remaining ninety percent is standard engineering work the team has done before. Smith and Reinertsen documented the same ratio in their book on accelerated product development [9]. Treating the program as if it were all innovation — or as if it were all standard development — produces predictable failure modes.

The lateralworks approach is to separate the two streams structurally. The innovation stream is isolated, dedicated, and protected. The conventional stream runs alongside on standard development practices. The two converge at integration.

Section 07 — continued

The 10 percent that matters

When the breakthrough work is not isolated, two failure modes appear. The first is fixation. The team and senior management focus collective attention on the difficult problem, allowing the easy ninety percent to slip. The "easy" work becomes the new critical path, which is invisible until the breakthrough is solved and management asks why the product is not ready. lateralworks has seen this pattern across semiconductor process development, storage systems firmware, and quantum communication programs. The diagnosis is the same in each case: the team that was assumed to be doing "easy" work was not given the urgency or the support it needed [3].

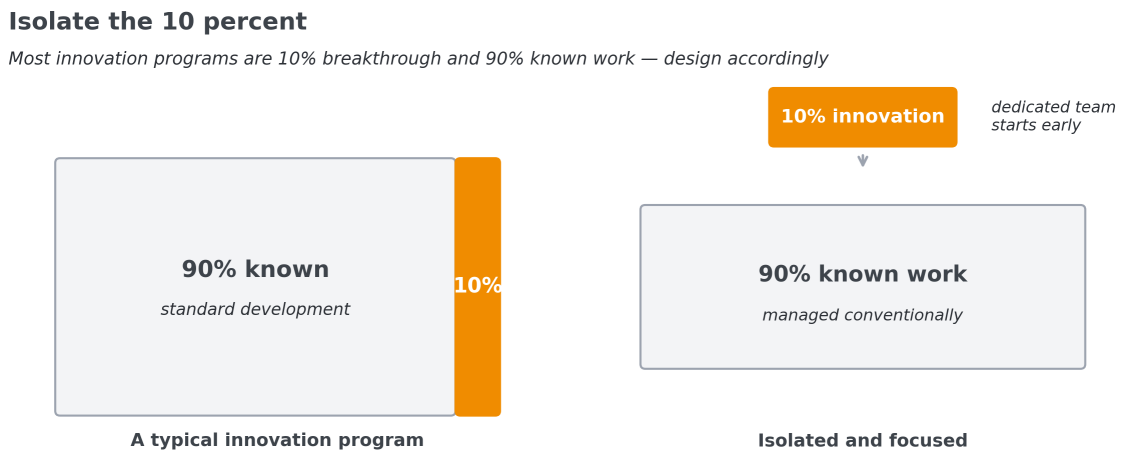

Figure 9. Most innovation programs are 10 percent breakthrough and 90 percent known engineering work. Isolating the 10 percent into a dedicated team — with an early start, dedicated resources, and protection from program-wide interrupts — is the structural pattern that wins.

The second failure mode is resource starvation. When the same engineers are expected to deliver both breakthrough and standard work, the standard work is allowed to interrupt the breakthrough work because the standard work has more concrete deliverables. The breakthrough work gets fragmented attention, multiplying cycle time. The remedy is staffing isolation: a dedicated team for the ten percent, with its own physical space if possible, its own cadence, and its own protection from the rest of the program's interrupts.

Start early; give it more time

The breakthrough team should start before the rest of the program. Conventional sequencing puts research and innovation alongside development, which means breakthrough work happens under launch-date pressure. Starting six to twelve months earlier converts a high-pressure breakthrough into a normal-pressure one, and gives the team the cycle count it needs to compress the schedule honestly. When the breakthrough delivers earlier than the rest of the program is ready, the resulting slack absorbs late-cycle

issues elsewhere — which is almost always where the schedule problems actually emerge.

A program with too many breakthroughs in flight has a different problem. Two is manageable. Five is not. lateralworks has seen programs that attempted to innovate on the IC design, the manufacturing process, the firmware architecture, and the package simultaneously. The program ground to a halt because every dependency required something else to break through first. The remedy is scope reduction: pick the one or two breakthroughs that are genuinely strategic and rely on proven technology for the rest. Christensen made the same observation in his work on disruption: organizations that try to innovate on every dimension simultaneously rarely innovate on any dimension at all [20].

08

Structure

Learning milestones

Conventional development uses feature milestones. Prototype demonstrated. Beta shipped. Production qualified. These milestones work when the path to the feature is known. They do not work for the ten percent of the program that is genuine breakthrough work, because there is no feature to demonstrate until something has been learned.

For breakthrough work, the milestone structure changes. Doneness is defined by what was learned at each stage rather than by what was built. Learning milestones measure progress toward a solution; feature milestones measure progress toward a product. Advanced programs need both, sequenced correctly.

Section 08 — continued

Doneness defined by learning

A learning milestone has the same structural role as a feature milestone but a different completion criterion. Where a feature milestone says "the prototype demonstrates functions A, B, and C," a learning milestone says "the root cause of mechanism X is understood and reproducible" or "the candidate set has been narrowed to three approaches with quantified trade-offs." The criterion is a piece of knowledge, not a piece of hardware or software.

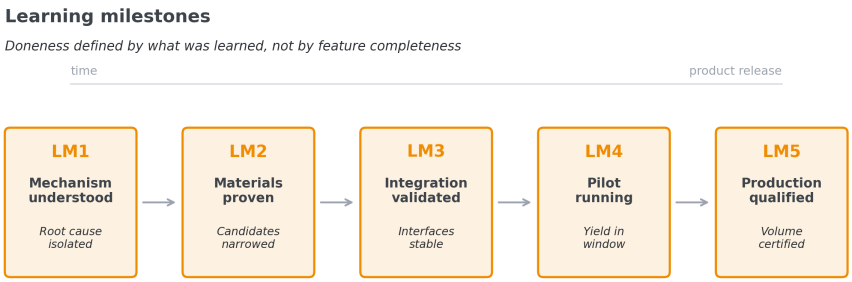

Figure 10. Learning milestones sequence the breakthrough work. Each milestone is defined by what the team will have learned, not by what the team will have built.

The structure works because it makes the learning visible. A traditional schedule that says "research — 6 months" tells management nothing about whether the research is on track. A learning-milestone schedule that says "by week eight, mechanism understood; by week sixteen, materials proven; by week twenty-four, integration validated" creates a sequence of checkpoints, each with its own pass/fail criterion. Slip in any milestone surfaces immediately, with enough lead time to act.

In one anonymized lateralworks engagement, a team faced a device-level failure where two components destroyed each other on first integration. The traditional schedule had marked "integration" as a single six-week task. After re-planning with learning milestones, the same work expanded into four milestones — mechanism understood, failure isolated, mitigation tested, integration retested — with explicit doneness criteria for each. The total time was the same, but the team and management could see weekly whether the work was converging. When the third milestone showed the mitigation was insufficient, the program pulled in a parallel approach four weeks earlier than it would have under the original schedule [3].

How to write doneness criteria. A learning milestone’s doneness criterion should be a sentence completing the phrase "We will know we are done when we can…" Examples: "…predict failure rate within ±10% across the parameter range," "…reproduce the result with three different operators," "…show the mechanism is independent of the substrate." Avoid criteria that describe activity rather than knowledge ("…have run 30 DOEs"). The point is what the team learned, not what the team did.

09

Playbook

Implementing learning cycles

The previous sections describe the components of the method. This section combines them into a sequence a program can adopt. The steps are ordered to deliver value in the order most teams need it: estimate first, then accelerate, then sustain. A program in pilot mode should be able to complete steps one through five inside four weeks, and steps six through nine inside the following quarter.

The playbook assumes a program that already has a defined product target and a rough scope of work. It does not address upstream choices about portfolio prioritization, market selection, or the choice of which technology to bet on; those are covered in the lateralworks portfolio papers [21, 22]. Within an active program, the steps below are the working sequence.

Section 09 — continued

Nine steps the team can adopt

1. Identify the breakthrough work. List the elements of the program that require genuine learning rather than execution of known practice. Aim for a list of three to seven items, no more. If the list runs longer than seven, the program is over-scoped and should be narrowed before continuing.

2. Classify each item easy, hard, or difficult. Convene the engineers closest to each item. Classify by consensus. Push back on optimism — "difficult" classifications cost less than missed schedules.

3. Calibrate cycle counts and durations. For each class, set the expected cycle count and typical cycle duration. Use the team's recent project history to calibrate, not industry averages. Validate the calibration by retro-predicting two or three recent efforts.

4. Compute the breakthrough estimate. Multiply cycle count by duration for each item. Sum across items, accounting for which items can run in parallel. Express the result as a range, not a point. Compare the range to the market window. Surface the gap explicitly.

5. Apply the two acceleration levers. For each breakthrough item, identify how concurrency can be added — typically by running two to four design-of-experiment paths in parallel rather than serially. Separately, identify how cycle time can be compressed — typically by better instrumentation, faster prototyping, or dedicated equipment. Cost the changes against the cost-of-delay [6, 17].

6. Isolate the breakthrough team. Staff the breakthrough work with a dedicated cross-functional team. Give them their own location if possible, their own cadence, and explicit protection from standard-program interrupts. Start them six to twelve months ahead of the rest of the program when feasible.

7. Replace feature milestones with learning milestones. For the breakthrough stream, define milestones by what the team will have learned, not by what the team will have built. Write explicit doneness criteria in the form "We will know we are done when we can…"

8. Adopt weekly refresh planning. Run a 60–90 minute refresh meeting each week. Detail the next four to six weeks; leave the rest as macro chunks. Roll the window forward as cycles complete. Track the gap between estimate and reality and adjust both the easy/hard/difficult calibration and the acceleration levers as needed [3].

9. Review trend, not status. In senior reviews, present the trend in expected delivery date across the past four to six refreshes rather than the current pass/fail status of recent cycles. Trending toward target is the leading indicator. Recent cycle pass/fail is noise.

What success looks like

A program running the full method has three visible signatures. The team can state, in one sentence, what the next learning cycle is intended to answer. Senior management can state, in one sentence, why the current delivery estimate is what it is. The trend line of expected delivery dates has narrowed week over week as the team has accumulated cycle data. Programs without these three signatures are running on hope; programs with them are running on method.

The method does not guarantee that breakthroughs arrive faster than physics permits. It does guarantee that the program will know, with operational accuracy, when it expects to ship and what changes would accelerate that date. In lateralworks engagement data, programs that adopt the full method routinely compress advanced-technology development to 60–70 percent of their original schedule, with prediction accuracy good enough to inform capital allocation and market timing decisions with confidence [3].

> **The shift in question.** The method changes the question senior management asks the team. "When will you finish?" becomes "What does this cycle teach us about when we will finish?" The first question invites the team to defend; the second invites the team to learn out loud. The shift is small in language and large in result.

Sources

References

[1] Smith, P. G., and Reinertsen, D. G. *Developing Products in Half the Time: New Rules, New Tools.* 2nd ed. Van Nostrand Reinhold, 1997.

[2] Smith, P. G., and Reinertsen, D. G. *Developing Products in Half the Time.* Van Nostrand Reinhold, 1991.

[3] lateralworks. "Engagement data across advanced-technology programs." Internal assessment database, 1988–2026.

[4] MacCormack, A., Verganti, R., and Iansiti, M. "Developing Products on ‘Internet Time’: The Anatomy of a Flexible Development Process." *Management Science,* vol. 47, no. 1, January 2001, pp. 133–150. https://doi.org/10.1287/mnsc.47.1.133.10663

[5] Iansiti, M., and MacCormack, A. D. "Living on Internet Time: Product Development at Netscape, Yahoo!, NetDynamics, and Microsoft." Harvard Business School Case 697-052, November 1996 (revised June 1999).

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

[7] Thomke, S. H. *Experimentation Matters: Unlocking the Potential of New Technologies for Innovation.* Harvard Business School Press, 2003.

[8] Thomke, S. H. *Experimentation Works: The Surprising Power of Business Experiments.* Harvard Business Review Press, 2020.

[9] Smith, P. G., and Reinertsen, D. G. "Shortening the Product Development Cycle." *Research-Technology Management,* vol. 35, no. 3, May–June 1992, pp. 44–49.

[10] Cusumano, M. A., and Yoffie, D. B. *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft.* Free Press, 1998.

[11] Edmondson, A. C. "Psychological Safety and Learning Behavior in Work Teams." *Administrative Science Quarterly,* vol. 44, no. 2, June 1999, pp. 350–383.

[12] Edmondson, A. C. *The Fearless Organization: Creating Psychological Safety in the Workplace for Learning, Innovation, and Growth.* Wiley, 2019.

[13] Shewhart, W. A. *Statistical Method from the Viewpoint of Quality Control.* Graduate School, U.S. Department of Agriculture, 1939.

[14] Deming, W. E. *Out of the Crisis.* MIT Center for Advanced Engineering Study, 1986.

[15] Ward, A. C., Liker, J. K., Cristiano, J. J., and Sobek, D. K. "The Second Toyota Paradox: How Delaying Decisions Can Make Better Cars Faster." *Sloan Management Review,* vol. 36, no. 3, Spring 1995, pp. 43–61.

[16] Sobek, D. K., Ward, A. C., and Liker, J. K. "Toyota’s Principles of Set-Based Concurrent Engineering." *Sloan Management Review,* vol. 40, no. 2, Winter 1999, pp. 67–83.

[17] House, C. H., and Price, R. L. "The Return Map: Tracking Product Teams." *Harvard Business Review,* January–February 1991, pp. 92–101.

[18] Edmondson, A. C. "The Competitive Imperative of Learning." *Harvard Business Review,* July–August 2008.

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

[20] Christensen, C. M. *The Innovator’s Dilemma: When New Technologies Cause Great Firms to Fail.* Harvard Business School Press, 1997.

[21] lateralworks. "Faster portfolios: AHP prioritization and starts control." White paper, 2026. https://lateralworks.com/papers

[22] lateralworks. "FTTM process maturity: a four-month migration playbook." White paper, 2026. https://lateralworks.com/papers

[23] Ries, E. *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses.* Crown Business, 2011.

[24] lateralworks. "Learning cycles." Ideas article, 2019. https://lateralworks.com/ideas/learning-cycles

[25] lateralworks. "Innovation can’t be predicted." Ideas article, 2019. https://lateralworks.com/ideas/innovation-cant-be-predicted

[26] lateralworks. "FTTM innovation." Ideas article, 2019. https://lateralworks.com/ideas/fttm-innovation

[27] lateralworks. "Do-it, try-it, fix-it cycles of learning." Ideas article, 2019. https://lateralworks.com/ideas/do-it-try-it-fix-it-cycles-of-learning

[28] lateralworks. "Fail fast." Ideas article, 2019. https://lateralworks.com/ideas/fail-fast

[29] Rodgers, T. J. "Five Minutes with TJ Rodgers, Founder and CEO, Cypress Semiconductor." Interview, *Embedded Computing Design,* 2014. https://embeddedcomputing.com

[30] Bohn, R. E. "Stop Fighting Fires." *Harvard Business Review,* July–August 2000, pp. 82–91.

[31] Goldratt, E. M. *Critical Chain.* North River Press, 1997.

[32] Cooper, R. G. "Stage-Gate Systems: A New Tool for Managing New Products." *Business Horizons,* vol. 33, no. 3, May–June 1990, pp. 44–54.