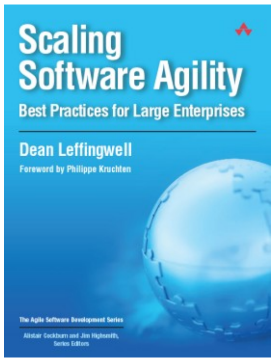
Leffingwell, LLC.

Whitepaper


Scaling Software Agility
Best Practices for Large Enterprises
Dean Leffingwell
Foreword by Philippe Kruchten

Systems of Systems and the Agile Release Train: An Agile Whitepaper

Dean Leffingwell

Abstract:
As agile development methods are applied to building larger and larger systems, it becomes necessary to anticipate, plan for, and manage dependencies among distributed agile teams. This necessitates more planning and better coordination of release cycles. In order to address this problem, we recommend an "agile release train" delivery model and metaphor which synchronizes teams as well as the maturity of their software assets to provide frequent and timely solution delivery to the marketplace. In so doing, we can achieve levels of agility, productivity, and quality far in excess of what we could accomplish with a more ad hoc, asynchronous model.

This paper is an excerpt from Dean's book, *Scaling Software Agility: Best Practices for Large Enterprises*, Addison Wesley, 2007 and is reproduced with the permission of the publisher.

Contents

Chapter 18

Systems of Systems and the Agile Release Train

When building large-scale systems of systems in a substantially agile manner, the practice of agility gets a little more complicated. But then, what doesn't?

Introduction

In earlier chapters, we spent time discussing complications that arise when systems get large enough to expand beyond the boundaries of what can be accomplished by one or two collocated, agile component teams. Indeed, many of the assumptions that the lighter weight methods have used to teach us agility may no longer apply.

In Chapter 16, Intentional Architecture, we looked at ways of organizing around the systems architecture so that local teams could operate in an almost completely agile and self-contained fashion, and we also illustrated how the interfaces between these teams would become increasingly more important as the scope of the system increased. In Chapter 17, Lean Requirements at Scale, we described a set of methods, including establishing and communicating the vision as well as defining and documenting common and mandated requirements (such as GUI standards, internationalization requirements, usability standards, regulatory and compliance standards, and system and performance standards) that are to be applied to the system as a whole and that provide an umbrella for the work that follows.

Also, in Part Two of the book, we described a set of best practices for the component teams that create an agile and reliable software production machine that we can now use to address ever-larger scale problems. Taken together, we now have the team skills, assets, and artifacts necessary to approach building systems of very large scale.

In this chapter, we look at agile planning and management constructs that are necessary to plan and build these larger-scale systems. And since we already illustrated that planning and agile are not mutually exclusive constructs, we extend what we've learned so far and do a little more planning than we have had to apply up until this point.

Specifically, as systems scale, it becomes necessary to anticipate, plan for, and manage dependencies among distributed teams of component developers. This necessitates more planning and typically somewhat longer release cycles (up to 3 or 4 months) to accomplish. In order to attack this problem, we use a component-based, "agile release train" delivery model and metaphor. In so doing, we can achieve levels of agility, productivity, and quality far in excess of what we could accomplish with our more traditional models.

An Agile Component Release Schedule

Prior to converting a team to agile practices, the time line model for a project might appear as in the figure 18–1.

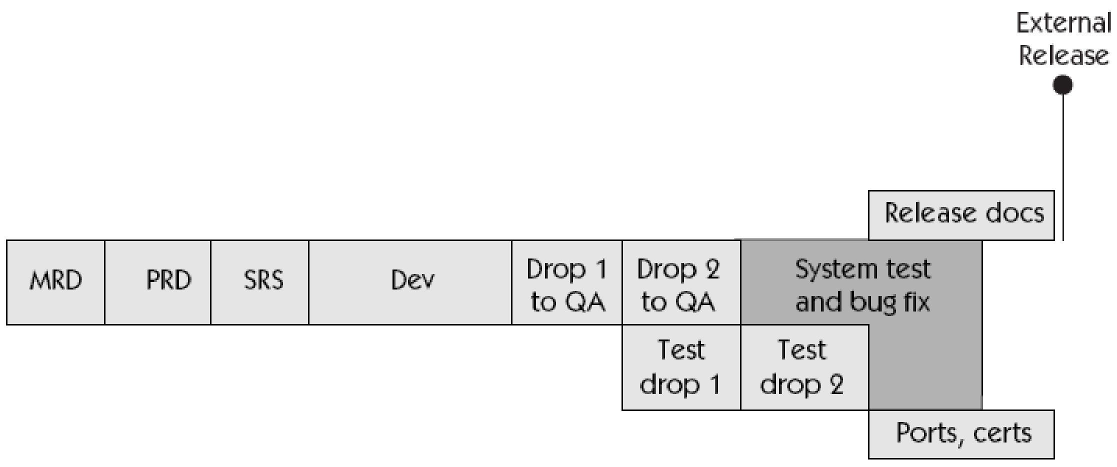


Figure 18–1 Pre-agile project time line

In this model, we see the waterfall activities providing a sequential view of requirements, then design, then development, then test, followed by any system-level testing activities plus porting and certification, user documents, and all other activities required to prepare the whole product for release. As we discussed, this model postpones most of the project risk until late in the time line, so we abandoned this model in favor of our new agile model, as illustrated in Figure 18–2.

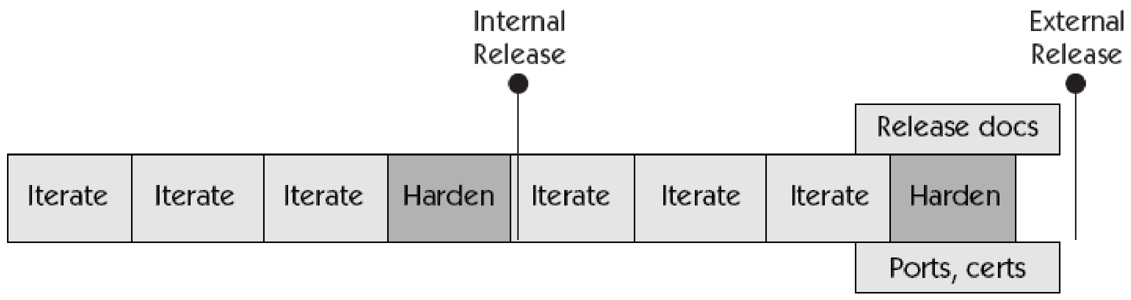


Figure 18–2 Agile component model development time line

In this figure, we see a typical agile component release schedule that provides internal or external releases of the component approximately every 60 days. (We applied the three iterations plus hardening pattern described in Chapter 13.) Iterations and internal releases provide constant visibility into the state of the project, and with some judicious, proactive, scope management, the time line is far more likely to be achieved.

Although this schedule is likely to be a substantial improvement over what the component team was able to do in the past and will produce better results, if we simply, blindly combine agile component release schedules at the system level, the release plan shown in Figure 18–3 emerges.

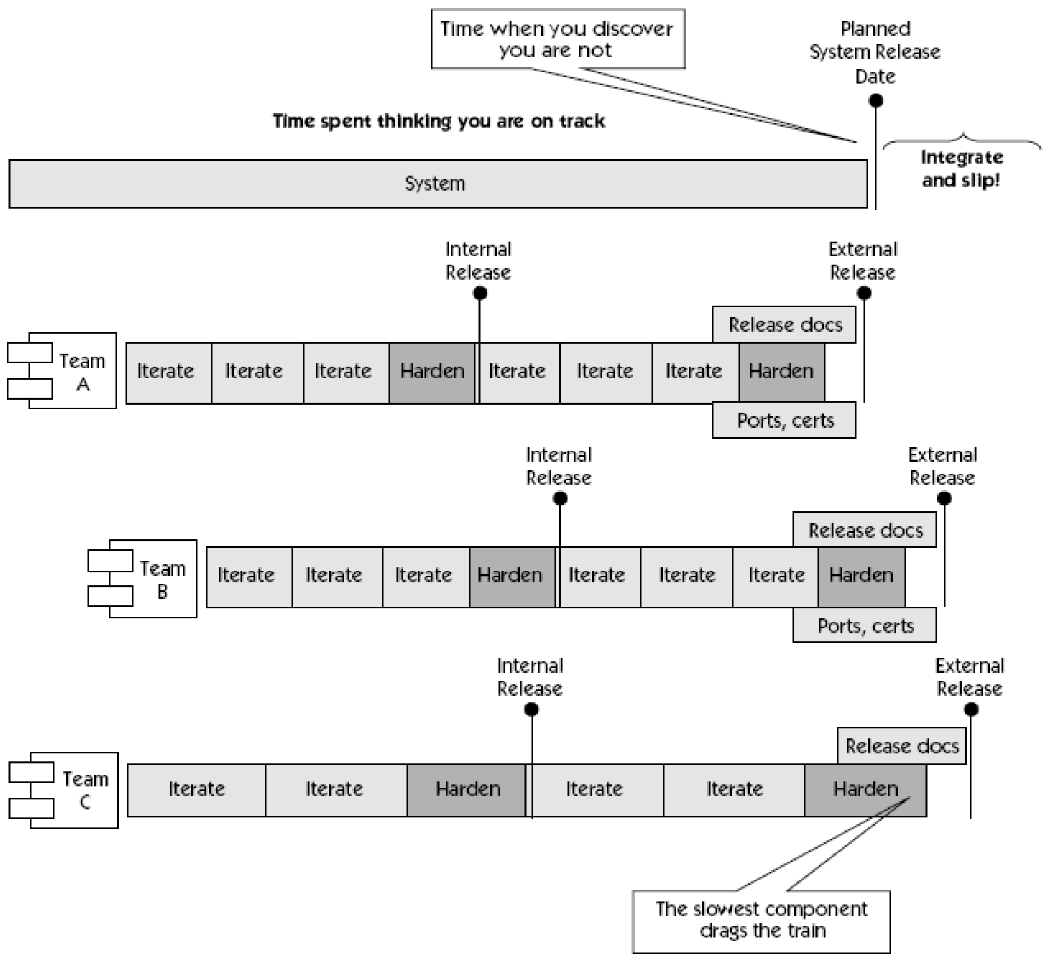


Figure 18–3 Unsynchronized agile release pattern

From this figure, you can see we have still not accomplished a reliable and synchronized release schedule. There are a number of reasons:

- The dreaded *system integration phase* is still largely with us, even though we have the component teams marching on a highly agile schedule! This is because the components themselves, while fully tested in their context, have not really been tested at the systems level. The net result is delayed risk discovery and likely schedule slips (and we thought we were agile).
- Forcing integration points to address this problem is compounded by the fact that one of the teams (Team C) operates on an entirely different cadence than the other teams, making integration asynchronous and awkward.
- The *slowest component (Team C again) drives the actual delivery date*, and you don't really know it until you get there.

In addition, decisions made near the release date are late-breaking compromises rather than early and conscious collaborations, reminding us very much of the problems of the traditional methods we were trying to avoid.

These problems are often compounded by large-scale architectural changes that were driven into the release at either the component or the system level. Since it is a new release after all, there are lots of new features, and some of those features require new infrastructure to be built. This late discovery process causes large-scale ripple effects back to the component teams. In the absence of fallback plans, with this model, the agile teams will still feel much of the pain of traditional methods.

Lessons Driving the Agile Train

We conclude from this experience that there must be a better way, and we need to adjust our system-level release strategies to become more agile as well. We also conclude that attempting to coordinate disparate component team milestones to an optimum release date is an over-constrained problem:

- There is no optimum date across all teams.
- The more teams there are, the more difficult it is to coordinate team milestones. For example, with distributed multinational teams, even holiday schedules do not align, so picking an optimum date for system release is likely impossible.
- Even if we worked out all the interdependencies (varying team calendars, etc) and we could determine an optimum date, it wouldn't match the external event calendar anyway (trade show dates, analyst briefing seasons, calendar-based events, and the like).

Instead, management must put a stake in the ground and mandate to the teams: *We will ship this often, and we will meet these dates*. However, in so doing, management must also recognize that specific decisions regarding individual component functionality must be left to the teams. Otherwise, everything is fixed: functionality, resources, and schedule. The result cannot be an agile process, and the teams are likely to fail to deliver.

Achieving effective release planning and meeting release dates for large-scale systems becomes the true art (agile release train) of the agile enterprise.

Principles of the Agile Release Train

To address the challenges of planning and scheduling, the agile enterprise must create a fixed set of rules that are imposed upon all the teams, and then leave the teams to figure out how to accomplish the mission. For the agile release train, these principles include the following:

- Frequent, periodic release dates for the system, platform, or solution are fixed and inviolate and known to all team members.
- Certain intermediate, global integration milestones are established and enforced.
- Even better, wherever possible, continuous system integration is practiced at the top system level as well as at the component level. This is not a trivial task, but it can be accomplished over time for most systems. As top system-level integration becomes the case, intermediate milestones typically evolve to be regular, internal releases available for customer preview, internal review, and system-level QA and testing.
- Constraining teams to the dates means that functionality for the components must be flexible.
- Certain infrastructure components, items such as common interfaces, system development kits, common installs, and the like, must typically track *ahead* of the component teams so they are available as necessary as the components advance.

- Each component supplier must evolve to a flexible new model: to be assured of meeting a date, a team typically needs to have both a primary plan and a fallback plan (each somewhere on the design continuum described in Chapter 17). In some cases, the fallback plan can be a simple as planning to ship the old version, if necessary, so long as the team tracks to any new interfaces and other common requirements that may be imposed on the release

The Agile Release Train

The result of this is the synchronized release train model is illustrated in Figure 18–4.

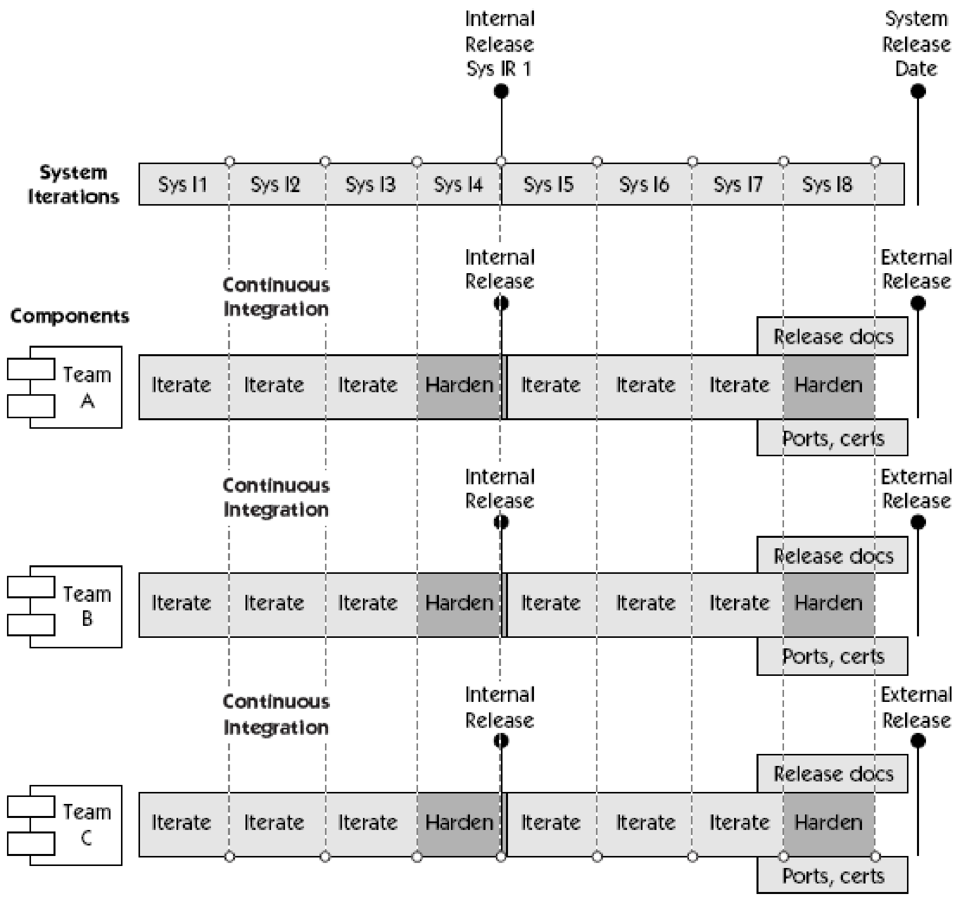


Figure 18–4 A synchronized agile release train

This Train Is Synchronized

In this model, all component teams are synchronized to the same iteration schedule. While this is not absolutely mandatory, it is extremely beneficial, and we might question why an enterprise could not simply mandate a synchronized iteration schedule. Again, in so doing, management is only mandating that the iterations be of one fixed length and that they all begin and end at the same time—management does not mandate what the iterations contain or how the team does its work.

In addition, continuous integration has been applied at the system level, *so the system now has iterations and internal releases available for customer assessment just like the components do*. This gives the team the objective evidence it needs to adjust system and component scope in real time and to shake out the issues that lie around the interfaces of the system. The risk is addressed early, rather than late, in the release cycle.

The slowest team no longer drives the train. Its relatively slower progress is detected early, scope is adjusted or resources are added, or in any case, *the team has the data it needs to take the appropriate action to get the train to the station on time*.

The Train Is Driven by the Vision, Themes, and End-to-End Use Cases

The input to the release train comes from the scalable release planning process described in Chapter 12. The vision is also an overriding presence to the planning process. It has been articulated in whatever agile manner the team so chooses, and it is known and available to all component teams. The vision input also includes the themes and highest level features for the system-level releases as well as any end-to-end use cases (i.e., those use cases that span the individual components to deliver end user value). Common system requirements (internationalization, accessibility, etc.) as well as the nonfunctional requirements (the "-ilities") are also known and factored into the individual component release plans.

Keeping the Train on the Tracks and on Schedule

Clearly, this train can neither design nor drive itself. As we described in Chapter 12, to achieve this model, companies organize around higher level management structures, often including a steering committee or release train management, as well as a Scrum of Scrums. But no matter the label, a team must exist to take responsibility for planning and managing the agile release train.

This team typically consists of some number of senior team leaders, development directors, QA personnel, and one or more system architects who together have the authority and responsibility to make sure the train delivers its goods on time. Responsibilities of this team (we'll call our team the Release Management Team, or RTM team for short) include:

- Setting the train schedule and all the integration milestones.
- Communicating the vision, including common requirements for the release (or the series of releases) to the team.
- Leading the release planning meeting that organizes the teams activities.

Quality members of the release team aggregate defects for system-level reporting to the RMT. They also continually concern themselves with overall system performance and reliability, as well as any blocks or issues associated with the build or integration problems. They also focus on all the other aspects of the "whole product solution," including items such as documentation, common install utilities, help systems, and any distribution, deployment, or support infrastructures necessary for successful delivery of the train to its users and stakeholders.

Measuring Course and Velocity

Typically, the RMT meets at least weekly to assess status, find and eliminate roadblocks, and adjust scope as necessary. Because a change to the plan of record is likely to affect some or all of the component teams, the RMT also has a responsibility to communicate any changes in plans or scope that it has determined to be necessary. The RMT should understand as part of the analysis that any changes in scope have a potential ripple effect through the teams.

At each internal release or other established milestone, the RMT team meets to analyze whether that particular milestone was met and whether the teams were able to deliver to the theme. It is also likely that some technical debt, such as necessary refactors, adjustments in scope, defect build-up, or deferred stories or features, is likely to have occurred. Based on this data, the RMT assesses the risks and reviews and revises the plan of record as necessary; it also considers any fallback plans that might be necessary to keep the train on schedule.

Observing System-Level Patterns

Because of the inherent measurability of the agile team process being applied, the release team will have access to constant real-time data regarding how the component teams and system-level efforts are progressing toward the goal. The RMT will likely soon notice that certain patterns begin to develop:

- Teams will exhibit varying velocities of conformance to the iteration plan. In other words, some teams will have relatively higher or lower rates of story achievement. This is a predictive measure, and the RMT can take such variances into account as it considers the current velocity and course of action.
- It is also likely that some teams start to run consistently late on release objectives. In that case, some coaching or additional, alternative resolution may be required. Perhaps scope management, adjustments to team members, additional resources, or realignment will help that team to succeed.

Note: The fact that a team is lagging does not mean it is necessarily a low-performing team. Infrastructure teams that have many interfaces and teams that take on significant technical risk, as well as teams that are good but overly optimistic, are often the teams that lag. Although lagging does not necessarily indicate a quality or performance measure for the team, it still must be addressed, or the train will not depart its destination on time.

Quality assurance plays a key role in this process as well. QA personnel will be collecting whatever metrics the teams create naturally, including the iteration metrics described in Chapter 15, and they will also be aggregating quality reports that summarize defects found across components and those attributed to the system level. These metrics give the RMT weekly objective status information about where the teams are. Fortunately, with agile, the team's agile metrics provide the raw data without much additional overhead.

Interdependencies Often Appear as Critical Blocks. Because the agile teams will have advanced their skills to the point where they can generally achieve the objectives that are within their control, and because they have accountability for specific components, they will naturally gravitate toward tasks that are under their control to assure their component is on schedule. However, interfaces among the teams, ambiguities among interface specifications, and assumptions that went into those interfaces, will likely create blocking issues that will appear on the release train tracks. These blocking issues must often be addressed by the RMT because they typically fall between component teams, affect multiple component teams, or even affect or impact teams outside the development organization.

Managing Interdependencies

As the individual component team's velocity and quality come to be understood, and as corrective actions are taken as necessary, and as the release date approaches, the RMT will often have to become actively involved in managing interdependencies. Much of this role will naturally fall to the system architects and senior product managers, but a few caveats should be applied:

1. Component teams are responsible and accountable for their mutual interfaces. In other words, responsibility for having interfaces work may not be "delegated up" to the system-level architects. Having mutual interfaces work is just one of the responsibilities of a component team, and this responsibility must be understood by the team.
2. In the extreme case, interfaces such as SDKs and APIs may need to be implemented as components themselves, and a dedicated team becomes responsible for implementing that abstraction layer and for communicating and testing interfaces to other components.

The Release Management Team can help with the management of these interdependencies by assuring that there is architectural review of these key interfaces in the planning and there is a focus on developing and testing these interfaces in the earliest iterations. Insisting on early and continuous integration is the most constructive move of all. For as continuous integration is accomplished at this highest system level, issues with the interfaces will expose themselves very quickly, and the component teams can adjust their course of action as necessary to minimize the risk and achieve the objectives.

Release Train Retrospective

In a manner similar to the iteration and release retrospectives that we described for component teams in Chapter 15, the RMT has the same set of responsibilities at the end of each release. A release train retrospective may not occur frequently, but it is an important element of regular reflection and adaptation, and the process is quite similar. The RMTs meet as a team and ask the following questions:

What did we deliver versus what we expected to deliver?
What debt did we incur?
What went well?
What needs improvement?
What one big thing could we do better next time?

The results of this larger scale retrospective will be of great interest to all team members, so results should be widely disseminated. The conclusions and takeaways can be folded into the next release planning session, and in this way an additional and continuous process improvement cycle works at the system of systems release level.