Making Architecture Visible to Improve Flow Management in Lean Software Development

Robert L. Nord and Ipek Ozkaya, Carnegie Mellon University
Raghvinder S. Sangwan, Pennsylvania State University

Release plans that give as much emphasis to architecturally significant tasks as to feature-based, high-priority functionality can achieve better outcomes by avoiding conditions that lead to wasted time and effort.

AN ARCHITECTURE EMBODIES those design decisions that influence a system’s quality attributes.\(^1\) Changes to an architecture involve modifying a system’s gross topology as well as its communication and coordination mechanisms. Therefore, any defects in an architecture (such as lack of desirable security, latency, or scalability) can necessitate enormous rework. An overly flexible architecture can lead to unnecessary work or overproduction. The time required to create such an architecture can also delay downstream activities.

Each software development paradigm treats architecture with a different focus. Phase-based software development methodologies, such as waterfall or Rational Unified Process allocate dedicated time and focus to different development activities, conducting them mostly in order (for example, requirements, analysis, architecture, development, and testing).\(^2\) Dedicating large chunks of time to architecture up front might not only delay downstream activities but could also easily lead to overproduction waste, where the effort spent might not pay off.\(^3\)

Iteration-based agile development methodologies, such as Scrum, work with predetermined two- to four-week sprints, each focused on feature-based, high-priority tasks.\(^4\) When there’s no up-front effort on architecting, developers must periodically conduct refactoring for necessary changes to accommodate new features. As the system grows, conditions will necessitate rearchitecting, which might require significant rework beyond the expected limits of refactoring. Sprint 0 aims to overcome this problem.\(^5\) Allocating architectural tasks to a sprint backlog, to allow architecture and feature development to progress in concert, has also been an increasingly recognized practice.\(^6\)

Although architecture in itself is important, prolonging the architectural work needed to support development can impede progress. A significant challenge is to determine what size increment of an architecture is best for a given iteration to manage the
Lean Software Development

In this article, we show how the flow management concept from lean software development can provide a framework for balancing the allocation of critical architectural tasks to development effort.

Release Planning: MS Lite Case Study

MS Lite is a hardware-based field system that automatically monitors and controls a building’s internal functions, such as heating, ventilation, air conditioning, access, and safety.

During its first year, the MS Lite project used an object-oriented analysis and design methodology. It captured requirements as use cases, which it grouped into functionally cohesive packages. It then distributed use case packages among different teams for development. Because the use cases weren’t fully elaborated, the team didn’t yet fully understand dependencies among the different use case packages. So, as the design emerged, so did new dependencies across packages, which created ad hoc communication streams among the different teams. Much to their chagrin, the teams found themselves in situations where they had to either wait for some yet-to-be-completed work by another team or perform duplicate work. Such interdependencies created conflicting solutions for cross-cutting concerns, such as dealing with variability in building automation devices and handling latency issues associated with the transporting events that such devices generated. As the design evolved and the teams continued to discover interdependencies, managing the workflow became so challenging that the project was stopped midyear.

In the second year, the project was undertaken from scratch, using architecture-centric methods. Building on use cases from the previous year, the project team made explicit the systemic quality of architecture can result in patched solutions leading to defects. Fixing these defects would lead to delay. Together, they all lead to wasted time and effort.

References


Architecture-related waste.

<table>
<thead>
<tr>
<th>Waste category</th>
<th>Relevance to software architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overproduction</td>
<td>Overproduction waste occurs when an excess of goods is produced. In software development, this can result from implementing low-priority or nice-to-have requirements, or creating an architecture for such requirements or for unforeseen futures without a sound basis.</td>
</tr>
<tr>
<td>Delay</td>
<td>Delay waste refers to the period of inactivity downstream in the process that occurs because an upstream activity doesn’t deliver on time. This type of waste occurs when an organization aims to collect all architecturally significant requirements and plans to architect for them, or aims to build utmost flexibility into the system, leading to overanalysis and prolonged time spent designing the architecture.</td>
</tr>
<tr>
<td>Defect</td>
<td>Defect waste occurs when the delivered software doesn’t meet the expected quality. Defects related to architecture can result in paramount rework when rearchitecting is needed.</td>
</tr>
</tbody>
</table>
attribute requirements that have the most significant influence on a system's architecture. The team used these requirements to design an architecture that reflected a better understanding of dependencies (both functional and systemic) among the MSLite system’s different elements before distributing these elements for development. Thus, the architecture became a mechanism for coordinating tasks among the different teams, resulting in a significantly improved workflow.

Our Experience

Being familiar with the MSLite project, we were motivated to use lean software development’s flow management concept (for more information, see the “Lean Development in Manufacturing” sidebar) as a framework for understanding how to balance the allocation of critical architectural tasks to a development effort. Figure 1a shows our adaptation of Don Reinertson’s visualization of optimum batch size to demonstrate the effect of architecture-related overproduction, delay, and defect waste on the overall project, and how the size of architecture increments affects the costs associated with waste. To demonstrate the batch size, Figure 1b shows how we tested our adaptation by looking at the amount of cumulative cost—or the running sum of the cost incurred (both implementation and rework) up to a given release—spent on architecting for MSLite data.

As Figure 1a illustrates, architecting in many smaller increments reduces the cost of delay that results from waiting for an entire architecture to be completed. However, rework is more costly, because it might involve rarchitecting. Having fewer large increments reduces the cost of rework, but at the expense of adding to the cost of delay for the larger architecture increment to be completed.

Delay costs are incurred because of waiting or delay waste. Rework costs are incurred because of defect waste as well as overproduction waste.

During its second year, the MSLite development path fell somewhere in the middle of the total cost curve. In trying to understand and improve the process (architecture and increment size), we modeled the delay and rework costs for this path in terms of the value of capabilities delivered over the total effort. To model the cost for the path, we looked at the MSLite project’s development plan, the software architecture’s module view, and its code (the DLLs).

Cumulative value is a running
sum of the value of delivered capabilities (all the user stories and architecturally significant acceptance test cases) up to a given release. Figure 1b depicts this on the y-axis. The pace with which value is delivered in each release is indicative of the cost of delay. For instance, there is a high cost of delay early on in the first release of the MSLite development path, path 3, where no value is delivered.

Flow Management
To see the influence of architecture on cost, we adapted path 3 using two strategies of development (modeled as paths 1 and 2 in Figure 1b) to realize the MSLite system’s requirements. Path 1 demonstrates release-planning strategies focused on delay-cost avoidance, and path 2 demonstrates rework-cost avoidance. These strategies represent the two extremes for decisions in iteration planning.

Avoiding delay cost. Path 1 reduces the cost of delay at the expense of rework, focusing on small architecture increments. This path uses the priority of stories and acceptance test cases to guide the work (referred to as enhancement agility). Using an agile approach with many short iterations delivering small increments keeps the delay cost low because there’s little waiting. However, architectural issues aren’t visible to the end user and are likely to be initially missed or ignored, resulting in a high rework cost later on, when they can no longer be avoided. Therefore, the total cost is quite high.

Figure 1b shows this. Path 1 starts out with a high throughput (a ratio of work in process [WIP] and cycle time) during the first two releases, then delivery tapers off as subsequent releases take longer. The WIP limits are fixed at three stories or acceptance test cases. The drag on throughput is an indication of high rework costs to deal with the growing complexity of dependencies. This shows the effect of small architectural increments (which Figure 1a also demonstrates toward the extreme left of the total cost curve); small architectural increments create rework waste. Path 1 in Figure 1b is typical of efforts where waste elimination and short-term value delivery aren’t balanced by consideration of the longer-term effects of the project’s rework cost.

Avoiding rework cost. Path 2 reduces rework at the expense of delay cost. It focuses on a few large architecture increments, using rework cost as a guide. Using a phase-based approach and carefully considering architectural dependencies can minimize or eliminate rework cost but at the expense of delaying activities downstream while the architecture is put in place. The total cost is therefore high.

Figure 1b demonstrates this, where path 2 starts out with low throughput of end-user value early on, owing to the delay cost while the team focuses on putting the architecture in place. Figure 1a also shows the effect of large architectural increments toward the extreme right of the total cost curve.

Once the architecture is in place, throughput does improve as the team settles into a rhythm of releasing high-value capabilities at regular intervals. The WIP limits change, however, because later releases must deal with an overloaded allocation of stories with the assumption that the architecture is ready. Although this might appear to achieve high throughput—a higher number of stories yields higher value for the release—it’s already on release
5, and there’s an added complexity of having to deal with the maximum throughput in the final release.

A comparison. Although we constructed paths 1 and 2, they’re grounded in the actual system’s building blocks (requirements and architectural elements) and can be seen as variations of path 3. We wanted to see how the architecture-centric development path of MSLite would compare to these paths and provide insights for the best strategy for allocating architectural tasks to a development effort. Table 1 compares these two extreme iteration-based agile and phase-based approaches on the basis of how they treat lean principles.

**Recommendations for Practitioners**

There’s no one-size-fits-all approach for achieving maximum throughput value with minimal resource costs. Focusing on flow management to analyze different approaches, however, can provide insight into how to improve management for both cycle time and WIP. One such insight involves improving the visibility of architecture-related aspects of a development project, as we demonstrated with the MSLite case study modeling different strategies for decision-making in iteration planning. Balancing the two opposing forces of delay-cost and rework-cost minimization will result in an improved outcome, as shown by the actual development path of the MSLite system.

The phase- or iteration-based approaches to software development typically don’t provide tools to visualize architecture’s role in managing developed features’ throughput. Therefore, we designed a Kanban board to elevate the criticality of the architecture and show how it supports or inhibits throughput. Kanban is a scheduling system that helps just-in-time delivery by emphasizing pulling work items from process queues with work limits. 12

Typical Kanban boards focus on feature, story, or task-level allocation of WIP limits—these limits are the numbers in the columns in Figure 2, mimicking the agile development context as shown in Figure 2a. Kanban is also used to support a phased-based approach where the WIP limits are organized according to an order of design, development, and test (see Figure 2b).

Our Kanban board design highlights architecture-related aspects to include infrastructure development, which is a critical aspect of improving throughput. There could be different versions of such a board involving parallel horizontal swim lanes or color coding, as demonstrated in Figures 2c and 2d, respectively.

![Figure 2](image-url)
A parallel swim lane for architecture-related tasks helped us achieve architecture development in concert with feature development to improve throughput, mainly by enforcing WIP limits on activities such as eliciting architecture-focused acceptance test cases, architecture prototyping, or rework to pay back architecture debt.

Enforcing technical debt WIP limits increases the visibility of rearchitecting for quality and also ensures that finding and fixing defects don’t come at the expense of overall system quality. Similarly, both architectural prototyping and pulling through architecturally significant requirements (captured as acceptance test cases) should be in balance with story development.

Our experience in experimenting with different release plans demonstrates how lean software development has the potential to merge architecturally significant tasks with feature-based high-priority functionality development, unlike agile software development methodologies, where sprints can create artificial boundaries and story slicing becomes increasingly challenging. By visualizing architecture-related tasks that contribute to feature throughput, including tasks that span multiple sprints, teams can achieve an effective flow-based development environment.

The following actions should be kept central to the development effort: first, capturing architecturally significant requirements as acceptance test cases enables them to be more easily visible on a backlog or a Kanban board. In a lean environment, enforcing WIP limits on these acceptance test cases assists with improving flow by managing the overproduction waste associated with overarchitecting. Second, when determining story priorities, considering the dependency of the stories on architecturally significant tasks enables the pulling of related stories earlier in development and assists with flow. Finally, in addition to monitoring story development, monitoring the changing quality of the system via its architecture, possibly captured as technical debt or rework-related defects, enables a response to the cost associated with rework-related waste.

Acknowledgments
This material is based on work funded and supported by the Department of Defense under contract number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Carnegie Mellon University makes no warranties of any kind, either expressed or
implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement. This material has been approved for public release and unlimited distribution except as restricted by copyright. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013 and 252.227-7013 Alternate I.

References