Big Agile: Enterprise Savior or Oxymoron?

Something strange is happening in enterprise software development—eager CIOs are launching “agile” projects with teams of thirty people or more devoted to a single product release. And why not? We know agile works well for small teams and small projects, and monster enterprise projects (like rolling out a new SAP financial solution to replace all your legacy systems worldwide) often require greater capabilities than a small team can provide. So why not scale up agile teams to maintain the cost and efficiency benefits of the agile process while accessing the necessary manpower to pursue complex global projects?

If it works, we’ll be enterprise heroes—ready to have our portraits enshrined in the corporate IT hall of fame. But what if agile only works when teams and projects stay relatively small? That’s the question most CIOs want answered before investing scarce time, energy, or resources chasing the “big agile” paradigm.

To get that answer, we turned to the only source we truly trust—cold, hard data from the QSM software projects database.

The Ground Rules

To find out whether agile delivers the same benefits when applied to larger endeavors, we analyzed roughly three hundred recently completed IT projects, half of which reported using agile methods and half of which did not.

Agile projects in the QSM database are those that were reported as such by the teams that developed and delivered them. The results of the study may be influenced by variability in how agile methods were applied, but that seems only fitting for a methodology that espouses the freedom to “adapt as you adopt.” We are actively collecting more agile projects. However, at this point, the sample size is still relatively small—approximately one hundred fifty agile projects. We’ll be interested to see how our initial observations hold up over time.

To measure the relative size of software projects, we looked at the number of source lines of code delivered when the system was put into production. Though agile projects frequently estimate using story points, ideal hours, or counts of stories instead of code, we can empirically determine the average code volume per story point from completed project data by dividing the delivered code by the number of story points. This works well for our purposes because we need a “ruler” for measuring the volume of work to be performed that’s independent of how the project is staffed.

In addition, we looked at time, effort, activity overlap, and productivity data for two high-level phases of each software project:

- The story writing, or requirements and design phase, encompasses requirements setting and high-level design and architecture. This includes the work in agile projects that is sometimes referred to as “getting to ready” and grooming the backlog.
- The code, test, and deliver phase includes low-level design, coding, unit testing, integration, and system testing that leads to deployment.

In the traditional waterfall method of development, a large proportion of the requirements and design work precedes coding, testing, and release packaging. Overlap between the two phases is minimal. Conversely, agile's iterative design cycles and just-in-time story detailing typically result in a great deal more overlap.

Hence, we were curious to study the proportion of time spent on requirements and design relative to the time spent on coding, testing, and packaging for delivery. In other words, how does time and effort spent on design work and story writing affect productivity?

Is Big Agile Effective or Not?

Enough process talk. Do agile methods translate well to large-scale software projects? You didn't really expect a simple yes or no answer, did you?

Any measure of effectiveness must first define what "success" looks like. Software effectiveness measures typically include one or more of these high-level management goals:

- Cost efficiency
- Schedule efficiency
- High productivity

An organization's definition of project success should align with its top priorities. Hence, organizations with significant cost and resource constraints should focus on completing projects inexpensively. Others may prioritize time-to-market or optimal productivity (finding the right balance between resources, schedule, and quality).

Let's examine each of these goals individually.

Success Case #1: Cost Efficiency

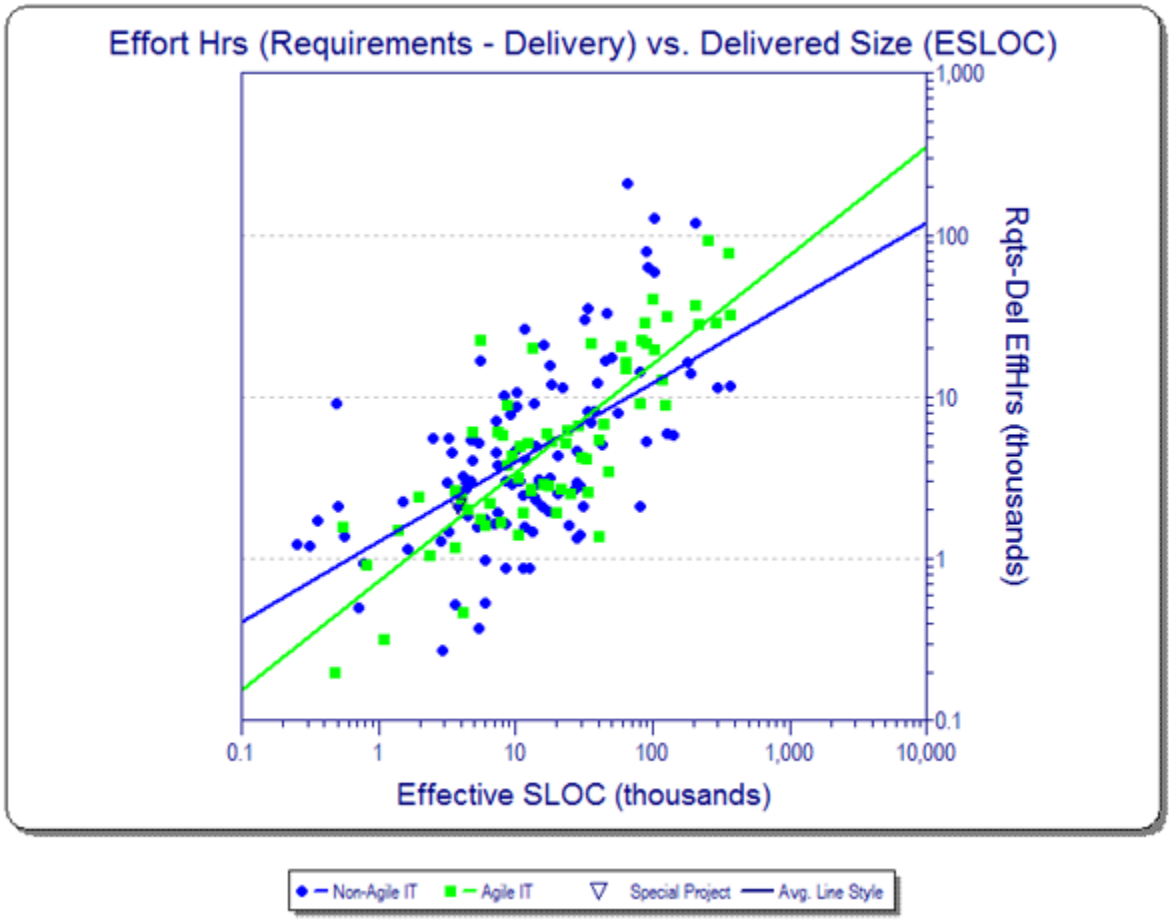

Figure 1. Cost Efficiency

To compare the cost efficiencies of numerous projects, we need to minimize the effects of varying labor rates. To that end, we viewed cost through the lens of effort hours expended (i.e., cost = effort * labor rate). Using this method, we can see that agile projects with fewer than thirty thousand source lines of code, or SLOC, are less costly than similarly sized nonagile projects. The trend seems to reverse itself in projects above the thirty thousand SLOC threshold, but in a subtle way.

The larger agile projects are fairly closely clustered; that is, for a particular size, there is not much variation in effort and cost. The larger nonagile projects, however, are either much less costly than their agile counterparts or quite a bit more costly. On average, large agile projects are a bit more costly, and this disparity seems to increase with size.

Bottom line: The agile cost advantage phases out at around thirty thousand SLOC. Above that threshold, agile projects may actually be more costly than traditional waterfall projects. If cost efficiency is your primary concern, both agile and nonagile projects should keep release sizes small.

Success Case #2: Schedule Efficiency

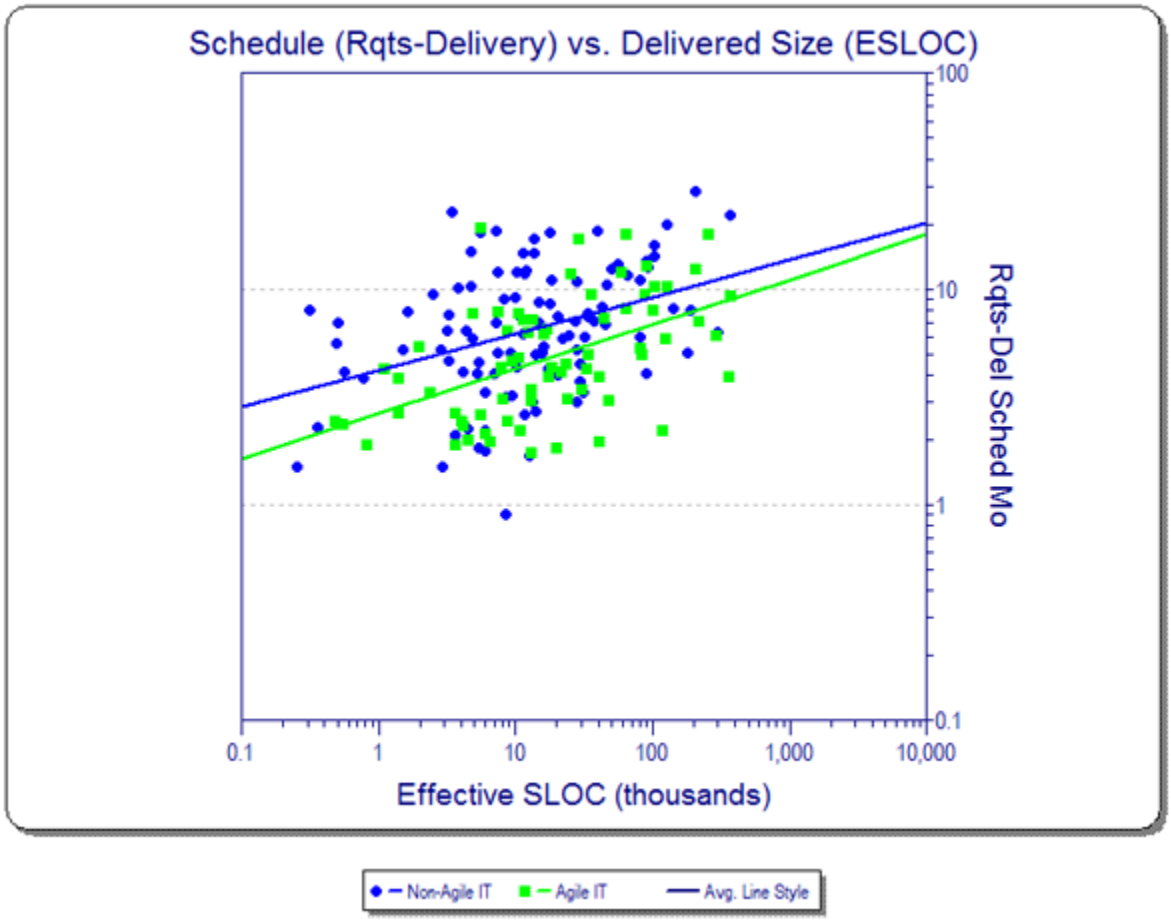

Figure 2. Schedule Efficiency

Our schedule analysis shows that time to market is consistently shorter for agile IT projects, but that agile schedule edge diminishes as the volume of delivered features increases.

Bottom line: If schedule is your primary concern, large agile projects are consistently more time-efficient than similarly sized nonagile projects, but the agile schedule advantage diminishes as project size grows.

Success Case #3: Balanced Productivity

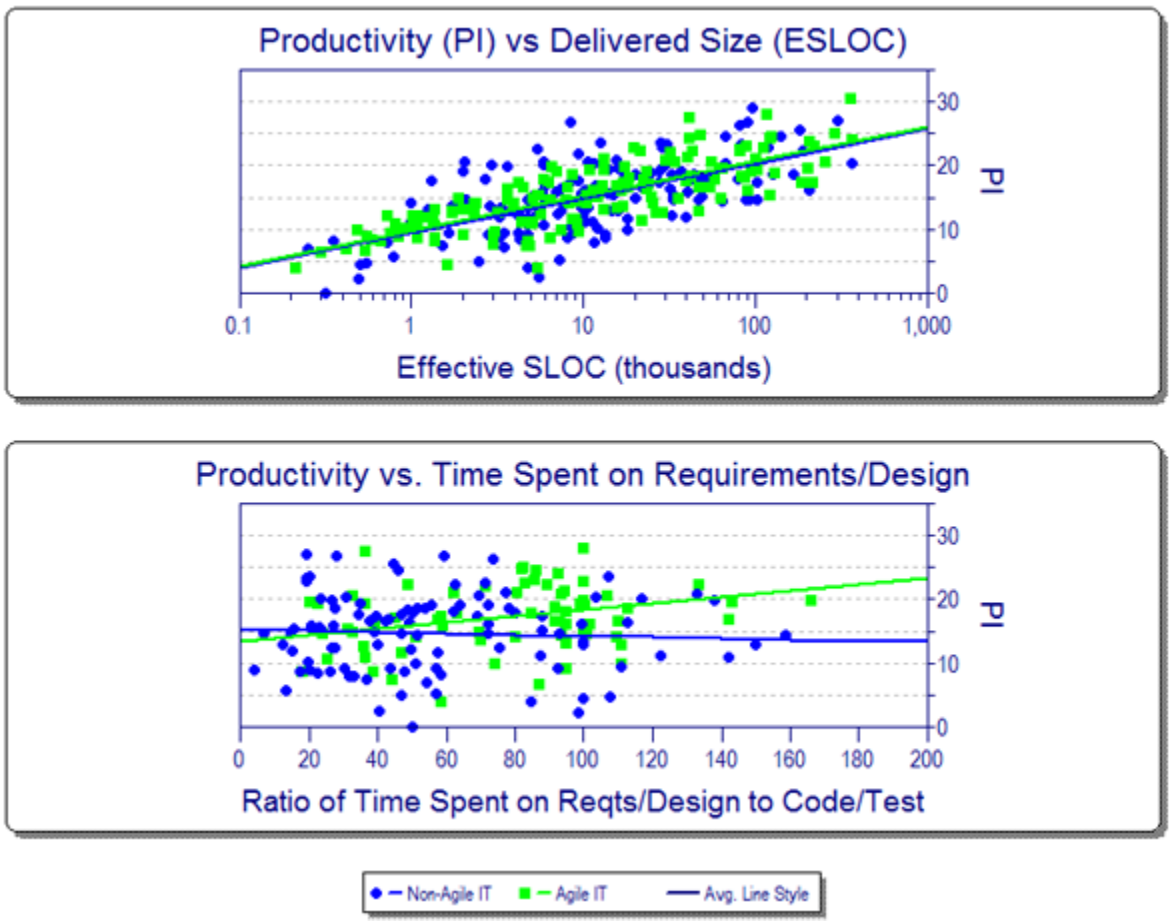

Figure 3. Balanced Productivity

To evaluate balanced schedule and cost productivity, we looked at a metric called the productivity index, or PI, which—unlike traditional productivity measures that examine resource or schedule efficiency, but not both—takes *both* time and cost into account.

According to the data, larger agile projects enjoy a modest productivity edge over nonagile projects, but again, the effect is quite subtle.

More interesting, perhaps, is what we see when plotting productivity against the proportion of time spent on design and story writing. Agile projects, it seems, become noticeably more productive as they spend a larger proportion of their time on requirements and design versus coding, testing, and packaging for delivery. This is consonant with findings in the agile community in general that taking extra time "getting to ready" and ensuring that user stories are well thought out and communicated is critical to the success of agile methods.

Bottom line: If process productivity is your primary consideration, large agile projects benefit from increased time spent on requirements setting and high-level design.

Wait, That's a Paradox, Isn't It?

Not necessarily. It’s true that one of the hallmarks of the agile method is spending less time up front on requirements and design. But “less” is a relative term. Although the time spent on requirements is more spread out throughout the project with agile methods, our data show that spending more time pays off in higher overall productivity.

Smaller projects are a more natural fit for “pure” agile—after all, most small projects were using small teams and lighter processes even before agile revolutionized software development. What we see from our data, however, is that as larger teams apply agile methods to larger projects, they are wisely adapting those “pure” agile precepts to the needs of larger, more complex systems, resulting in a tempered version of agile.

What’s most remarkable about this productivity chart, however, is that for nonagile projects, spending additional time in the design phase does not appear to boost productivity at all.

Agile methods, it seems, help teams apply that additional time and effort more effectively—working smarter, not harder.

Hence, big companies can benefit from constructing their software in an “agile-like,” iterative fashion with frequent reprioritization of features and functionality, as long as their requirements and design work is sufficiently robust. In fact, this is precisely what a growing number of industry experts such as Dean Leffingwell, Steve McConnell, and Scott Ambler are recommending.

Perhaps “big agile” is neither a savior nor an oxymoron; it’s simply a compromise using the best of new methods and tried-and-true techniques. After all, one of the agile tenets is allowing human judgment to trump rigid process guidelines.