Extracted from:

The Nature of Software Development

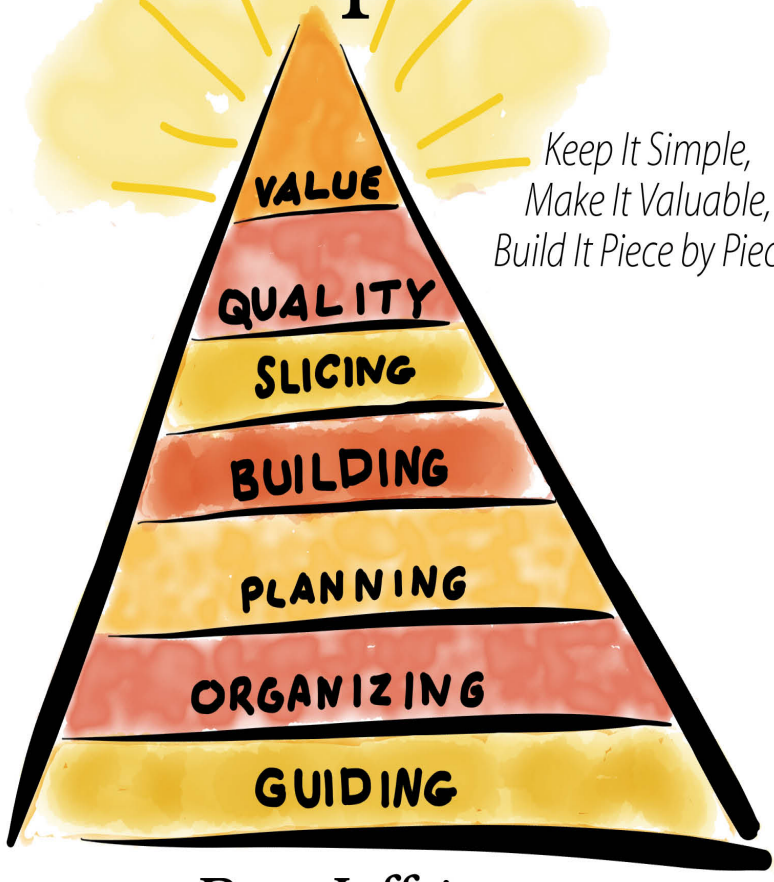
Keep It Simple,
Make It Valuable,
Build It Piece by Piece

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Nature of Software Development



Keep It Simple, Make It Valuable, Build It Piece by Piece

Ron Jeffries

edited by Michael Swaine

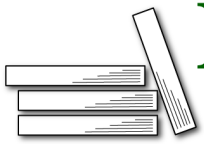
The Nature of Software Development

Keep It Simple,
Make It Valuable,
Build It Piece by Piece

Ron Jeffries

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

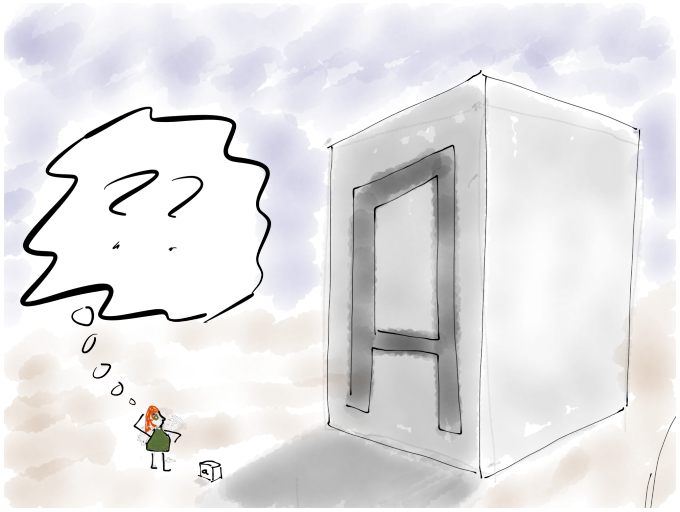
Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-37-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February, 2015



You don't need to "scale" Agile. You just need to do it.

CHAPTER 21

Scaling Agile

There's a lot of interest in "scaling" Agile these days, and it has become big business. Large companies have heard the clarion call of the Agile Buzzword, and just as they did with past good ideas like Six Sigma and TQM, now they want to go Agile. It has become the thing to do. But they're large companies. So, naturally, they think they need to scale.

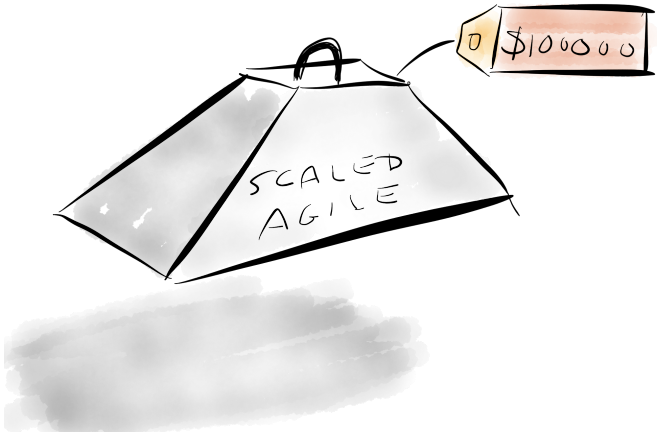
It turns out that in most cases, they're wrong. They don't need to scale. They need to do plain old simple Agile software development.

Scaling Agile is good business for scaling vendors. It's not necessarily good advice for you.

Scaling Agile has become a good business to be in, because people think they need it. There has always been a decent market for scaling Agile, so there have always been some contending approaches to doing it. Now, with the market for Big Agile growing, there are even more.

I'll leave it to you to look those up and choose among them if you must. What I'd like to do here is to suggest that—with one possible exception—these approaches are misguided.

That's not to say that large-scale Agile won't be successful; very likely it will be. It will be successful in the sense that large companies will buy scaling products and ideas, and consultants and training companies will enrich themselves selling what these large companies want.



As the Rolling Stones remind us, you can't always get what you want.

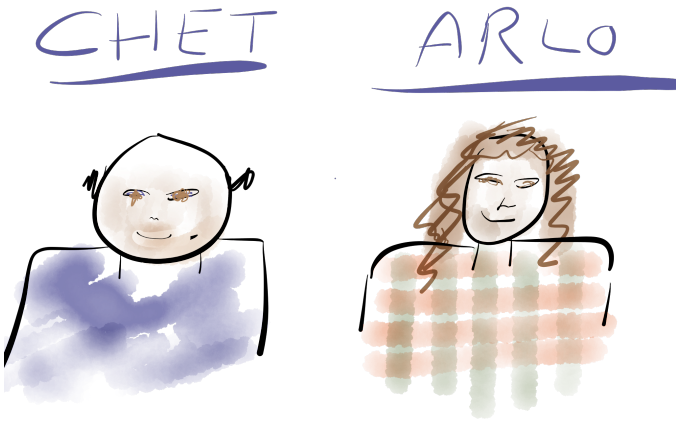
Unfortunately, contrary to the song, a big company can always get what it wants—in this case lots of expensive training in some heavy approach that touts itself as Scaled Agile. And they'll get some benefit, certainly. Any attention to improvement is usually better than no attention at all. And to the extent that these various approaches include some real Agile ideas, organizations will get some of that as well.

I'm here to talk about what the song offers: I'm here to talk about what you need. What does a large company really need to know about applying Agile ideas throughout?

Agile is simple—it just isn't easy.

Agile is quite simple. The most popular Agile approach, Scrum, has just three roles, a handful of activities, and one major artifact: running tested software.

That doesn't mean Agile is easy. It's still hard to decide what product would be desirable, and it's still hard to write software that does what is asked for. It is, however, quite simple. Simplicity is the essence of what makes up Agility. So if Agile is simple, what about so-called scaled Agile?

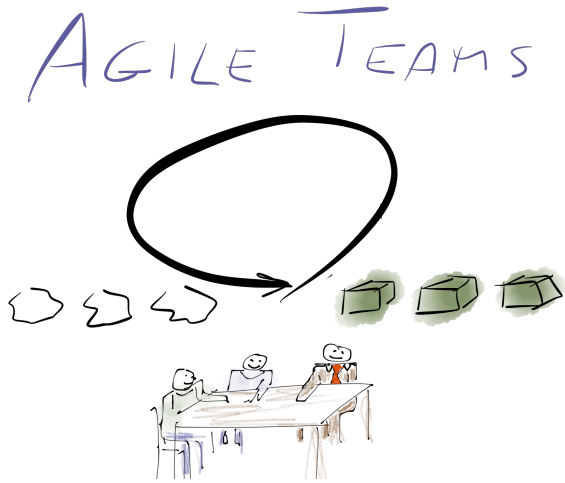


Scaled Agile must be simple—or it isn't Agile.

Chet Hendrickson points out that since Agile is simple, a scaled version of Agile should also be that simple, or even simpler. Otherwise, it will no longer be Agile. We should look with great suspicion at a so-called "Scaled Agile" approach that is complex.

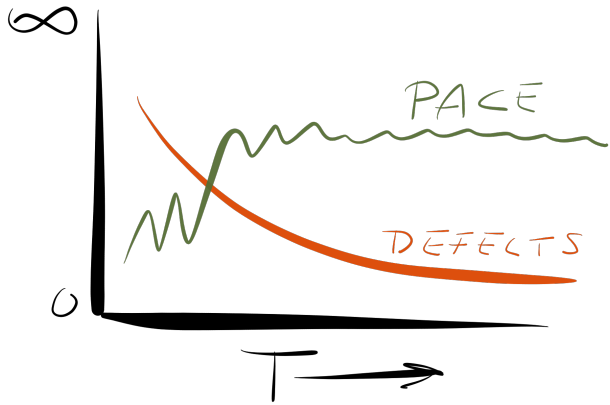
Along the same lines, Arlo Belshee suggests that if all your development teams have become fluent in Agile software development, scaling is not a problem. If all your teams can slice stories small, select a number that they can accomplish in a Sprint (or otherwise within time estimates), and deliver integrated software that is free of defects, then "scaling" should be easy. Diana Larsen and Jim Shore, originators of the notion of fluency in this context, make similar points.

Let's explore this. Agile is simple (but not easy). If your individual teams could really execute software development in the Agile style, might "Scaling Agile" be easy?



If your teams are truly Agile…

Agile teams work daily with their business-side associates (Agile Manifesto Principle 4). They deliver working software frequently, every couple of weeks (Principle 3). They measure themselves with working software (Principle 7), work in a sustainable fashion (Principle 8), and pay constant attention to technical excellence and good design (Principle 9). And so on.

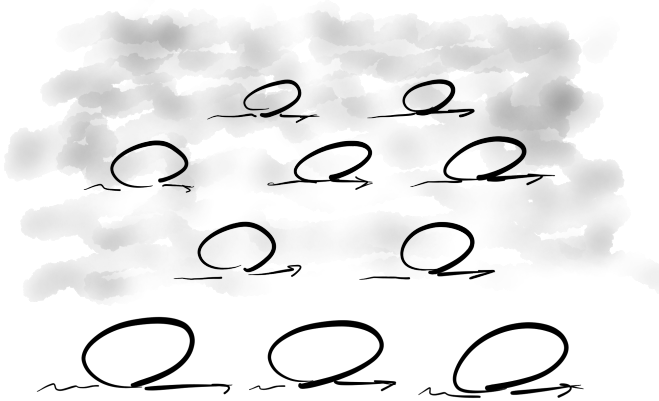


…I mean truly Agile…

Fluent Agile teams, after just a little jostling when they start up, produce a consistent flow of features, and they drive defects down to levels far below what the same team accomplished before they became fluent in Agile.

Fluent Agile teams are visibly Agile, visibly fluent. They get things truly done, at a consistent and predictable pace. If your teams are up to that…

…you might already be done.

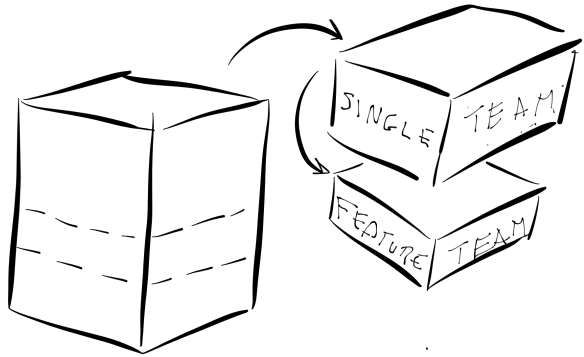


So there you are. All your teams are capable of producing working software, every two weeks, working daily with the business-side people who describe what the team needs to build.

You might be done scaling Agile, if everything your organization builds can be built by a single Agile team.

Really. Think about this for a moment. If everything you do could be built by a single small team, scaling Agile comes down to having each team learn to build in the Agile fashion, then hooking them up with a business-side person to guide what they build.

Done. Agile Scaled. No extra work beyond the basics. The basics are hard enough of course. We've explored that elsewhere in the book. But there's no big corporate rollout/transition/Enterprise Agile that you need to do.

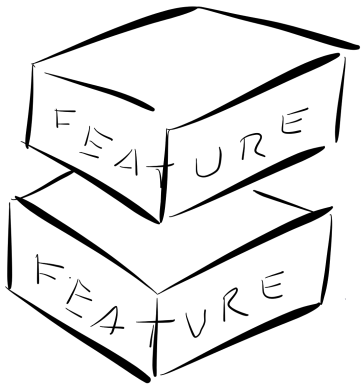


What if you want more than one team can do?

A single Agile team that can really do this stuff produces multiple features every couple of weeks. It's not easy to keep even one team working at capacity: you have to have a lot of product ideas to do it. But maybe you have a huge product, like a word processor or some graphics program for editing photographs. You feel there is enough work there to keep multiple teams busy.

Well, first of all, prove it. Get a single team working on your product up to Agile quality. Then look at the rate at which they deliver features. See if you really need more features than that. Odds are, you won't: your customers probably can't absorb new capability faster than a single team can deliver. But maybe there is enough work to keep more than one team busy.

Aha! Now we'll have to scale Agile…won't we?

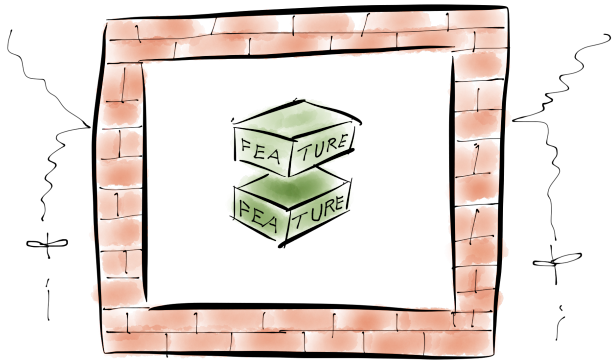


Feature teams

Well, maybe not. Way back in the last century, the idea of the feature team was devised. A feature team is a small team whose job it is to deliver features into a product. To get more features, you add more feature teams, all delivering software into a single product. Want more features per unit time? Add another feature team, get more features.

There's not much involved in scaling this way, is there? If every team knows how to do what a real Agile team knows how to do, you can add feature teams, and any product made of features can go as fast as you want.

Aren't we skipping something? How do those teams coordinate? Now we've got multiple teams doing features. How can they avoid stepping on each other's toes?



Agile teams coordinate using tests.

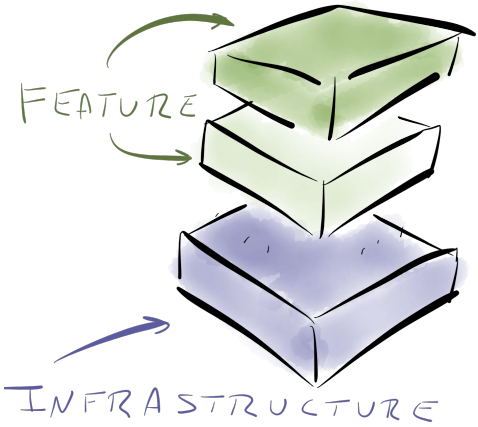
Remember that Agile teams do a large number of small features every two weeks. A single team can easily do fifteen or twenty such features in a two-week iteration. How do they manage not to get in each other's way?

It turns out to be simple. Fluent Agile teams build a growing container of automated checks, using acceptance test-driven development and test-driven development. These checks help teams know when they have completed a feature. However, they also serve as a growing collection of regression checks that ensure all the features built keep on working.

If we're using multiple feature teams, it works the same way. Each team, every time it builds a new small feature, adds that feature, with its automated checks, into the common codebase. All the teams do this daily, just as a single team would. They keep all the checks running, all the time. If from time to time a team tries to check something in and tests fail, they fix the problem before checking in so that the current codebase always runs all the checks.

Might there be a conflict between things done across teams? Possibly, and if that happens, the teams coordinate to see what happened. But the general practice is quite simple: if the checks were running before you put your change in, and they're not running after you put your change in, your change broke something. You find that something, and fix it, so that all the checks run—yours plus all the historical ones.

Agile teams do this as a matter of course. They learn to do smaller and smaller releases. When they use small releases, the chance that they break something is very small. When (rarely) they do break something, it's easy to find the issue, because only a small amount of code has been added or changed.



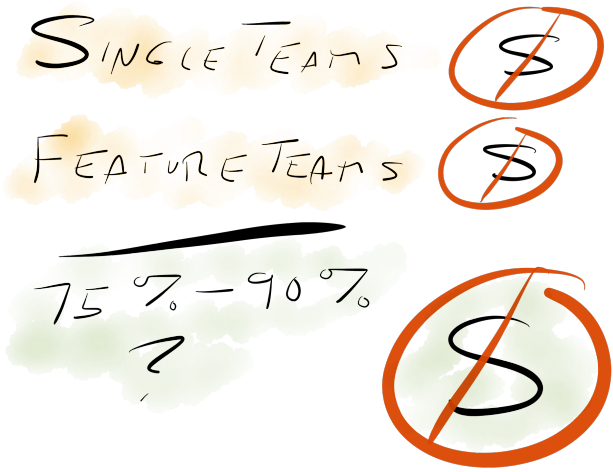
OK, feature teams, but what about infrastructure?

If your product is really big enough to use multiple feature teams, they'll be relying on some common infrastructure. What about changes to that?

Same way. Agile teams change their infrastructure as needed. They do so freely, every couple of weeks, by supporting their changes with automated checks. Feature teams can do the same thing, with each team making the changes it needs, adding checks to the pool, and checking in code frequently.

Will you need a specialized infrastructure team? Quite often, if you're fluent in Agile, you won't. Specialist teams very often dissolve in an Agile situation. But if you do choose to have such a team, and they're Agile, they can smoothly produce infrastructure changes, supported by automated checks, in support of multiple feature teams. I recommend letting your feature teams handle infrastructure changes, coordinating among themselves as needed. But if you do choose to have a specialized team for infrastructure, despite that advice, there's still no need for special scaling.

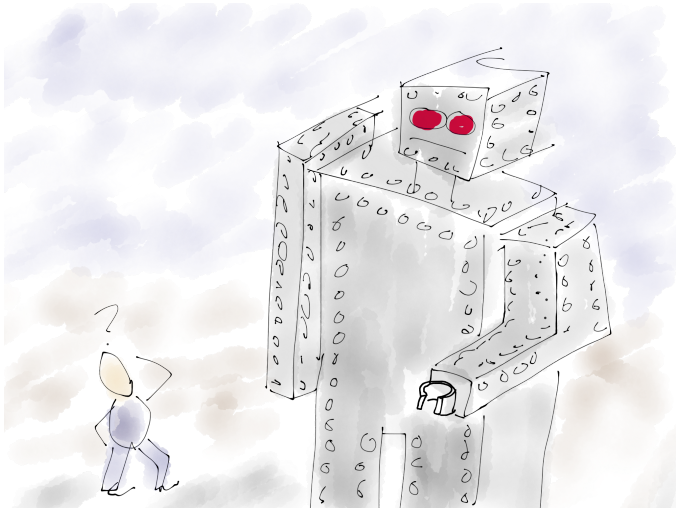
Remember, it's unlikely that you'll need feature teams at all if your individual teams can do Agile. But if you do, you won't need any special infrastructure to have feature teams—you just need multiple empowered teams who can, and will, coordinate among themselves as needed.



So far, so good

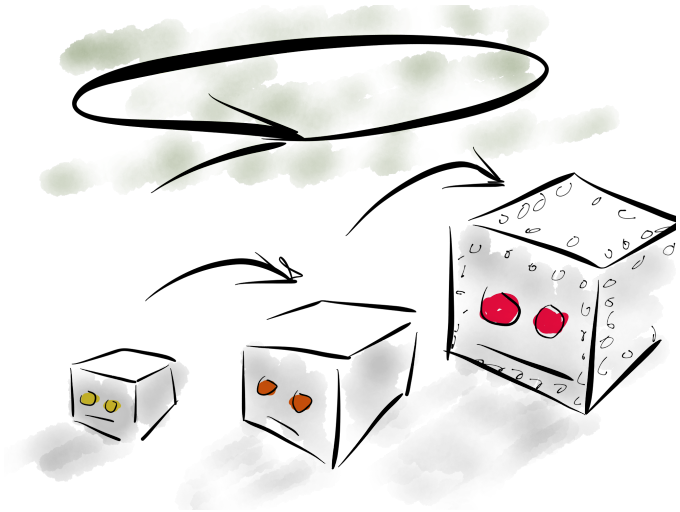
A company whose work can be done by a single team does not need anything special to scale Agile. A company with a need for more features than a single team can handle can build feature teams, and they won't need anything else to scale their Agile process.

In most organizations I've seen, the majority of the work is done by single teams already. In a few, I have seen a product that is integrated enough, and large enough, where feature teams might be needed. What else is there?



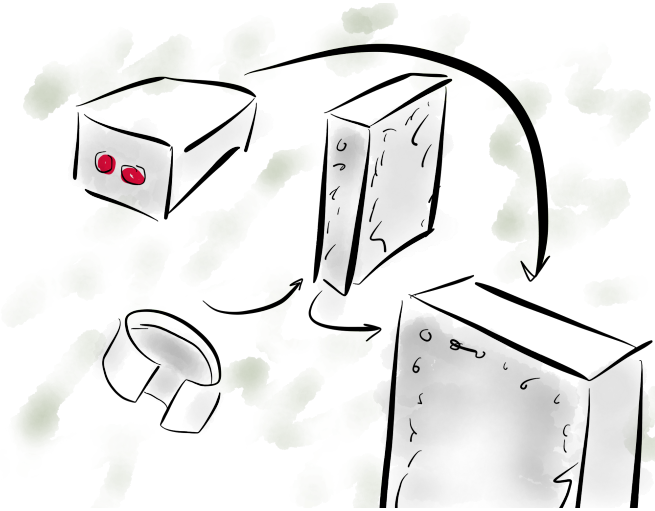
Giant efforts

Some companies undertake truly large efforts, with hundreds of developers, perhaps even thousands, all working on one thing. If you're not in a company like that, maybe all you need is to get your individual teams able to operate in an Agile fashion. You could stop reading now, or jump to the conclusion. But you're probably wondering what to do about giant efforts.



First, grow the giant incrementally.

If you're starting a giant effort, even one built on existing architecture, the standard Agile approach works. Start with a single team. Build it larger and larger. Build and extend infrastructure as you go. Add feature teams as you need them.



Finally, divide the giant, mostly along feature teams.

Even in giant efforts, it turns out that almost everything being done is being done by single teams. We already know how to do those: standard Agile. Just do that.

Even in giant efforts, a few efforts can be improved by adding more working teams. Do that, and operate them like feature teams. Standard Agile. Just do that.

What's left? Is there really something, somewhere, that needs more than one team, and that can't be divided up into smaller efforts that can be done in Agile fashion?

In most cases, I doubt it. I don't think there are giant efforts that are truly irreducible. If there are, no one knows how to do them, Agile or not. The very essence of putting lots of people on an effort is to divide up the work. If we do not know how to divide up the work, adding people will not help.

If we do know how to divide up the work, then, almost always, the bulk of the work can be done using standard Agile. Is there enough left to require a complex approach to scaling? Perhaps. Wait and see; that's my advice.

Bottom line

If your individual teams cannot work in an Agile fashion, then clearly you're not ready to "transition" your company or to "scale" Agile. You don't want to transition to something you can't do, and you don't want to scale something that doesn't work.

First, start creating teams that are very capable of doing Agile.

Then, give them the most important, most valuable work to do that your organization can come up with. And stand back.

Keep creating Agile teams, organized by features where possible. You may find that you have little need to scale Agile. More likely, you just have to do it.