# DevOps and the Division of Labor

Gene Callahan

NYU Tandon School of Engineering

The increased productivity brought about by the division of labor is a key insight of economics. This division reached in an extreme in the assembly lines of the 20th century, places where humans were asked not to think but to serve as capital goods, as flexible parts of a machine. The DevOps call to “break down silos” represents a reversal of that trend towards ever greater specialization. What is causing that reversal, and why is it happening now?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The benefits of the division of labor were, of course, recognized at least as far back as Plato and Xenophon. Adam Smith famously expounded upon them in *The Wealth of Nations*. In the early 20th century, this method of increasing productivity was pushed to its limits. Tasks were broken down to the extent that workers with minimal skills could be assigned simple, highly repetitive actions, and perform them with almost no knowledge of what anyone else on the assembly line was up to.

Although this led to higher productivity of standardized products, the disadvantages of extending the division of labor to this extent were not overlooked. Karl Marx noted that the extensive division of labor alienated the worker from the product he was producing: someone who spends all day tightening a particular lug nut may be little able to associate what they do with "making a car." But even Adam Smith, typically understood as a great proponent of the division of labor, commented that years at a simple, assembly-line task could make a person “as stupid and ignorant as it is possible to become for a human creature to become.”

Smith is pointing out a *general* problem with the extensive division of labor, but there is a much more *particular* problem, which only came to prominence in the recent days of increasing automation and increasing demand for innovative and customized products: the sort of mindless, production-line division of tasks common in mid-20th-century factories created a workforce downright *discouraged* from thinking about how their work fit into the production process as a whole, or how innovations in other parts they did not directly make might affect their own task. Such a holistic view was only supposed to be required of the engineers who designed new products or who designed the factory processes that would produce those new products. As in a socialist economy, all knowledge about the product and the production process would be concentrated at the top of a pyramid of work, and those below the peak were to just mindlessly follow the orders of those knowledge commissars.

A problem with this approach is that as products become more complicated and the pace of innovation increases, no single mind, or even a small group of minds, is capable of grasping all of the interconnections between the different parts of a complex product, and thus, cannot foresee how an innovation supposedly concerning only one part will actually have ripple effects on many other apparently separate production tasks. This fact was realized quite early at Toyota.

As important as these ideas were in factory production, their importance is even greater in the world of software development, where production is *always* production of a novel product: otherwise, one would simply buy or rent an existing software product, which is almost always a lower cost venture than "rolling your own."

In such an environment, it is simply not possible to assign the "workers" (programmers) a simple, repetitive task, and expect them to achieve decent results without at least some understanding of the overall product design, as well as an understanding of how their particular "part" integrates with the other parts of the product as a whole. In such a situation, worker obedience no longer "works." A manager cannot tell a software engineer working on a product of even moderate complexity to just follow the manager's orders: the programmer can bring production to a halt simply by asking, "OK, what line of code should I write next?"

But further: no knowledge worker producing an even moderately complex product can do his work properly without his understanding of his part in the production process evolving in continuous interaction with the evolving understanding of all of the other knowledge workers involved in the product: one such worker gaining a better understanding of the nature of her component simply *must* convey that understanding to all other workers upon whom the changes in her component have an impact, and that set of workers typically encompasses *everyone* working on the product.

The various aspects of Agile / Lean / DevOps production follow from the nature of knowledge workers cooperating to create innovative products. Programmers cannot do their jobs in isolation: thus, *continuous integration*. Testers cannot test successfully unless they are part of the production process from day one: thus, *continuous testing*. Operations cannot successfully deploy constantly evolving products unless deployment itself becomes a software product: thus, *software as infrastructure*. The "business" stakeholders in the product cannot ensure the product is really meeting business needs unless they are continually engaged in the development process: thus *continual interaction* between the engineers and the "business people." How new versions of a piece of software impact the end users cannot be determined without continual feedback from those users: thus, *incremental development*, *continuous deployment*, and *continuous monitoring*.