Let's talk about the two most basic development paradigms:

Waterfall and Agile.

(Slide 2)

Waterfall is the old-school way of building things. It has origins in manufacturing and heavy industry, and was once a standard for the Department of Defense, and it definitely shows; each stage depends utterly on the one before it, which means there is no opportunity for parallel efforts, and there is no room for failure. The end result of a waterfall development cycle is a single large deliverable, a complete product ready to ship. For a long time, software was engineered this way, and for good reason; without the internet, the best way to ship software is as a complete unit (everyone remembers giant stacks of floppy disks to install something). But it has one glaring flaw: no customer feedback during development, which means no rapid iteration.

(Slide 3)

Agile is the “new” way of doing things. It's actually old (hence the quotation marks), but became all the rage as a direct result of the internet revolution with the release of the Agile Manifesto in 2001. With Agile, you no longer need to ship full, complete units. Now we can ship small units very quickly, get fast feedback, adapt, and ship again; and again, and again. This is true in nearly all software today, but is especially and emphatically true in the web. The key to this paradigm is rapid incremental improvement. Get something out, let teams work in parallel, continuously improve the result, and scale to something great.

(Slide 4)

With Agile, we can "Move fast and break things," as is fun to say.

Since Agile does not depend on a single, large-scale deliverable, as with Waterfall, there is a heavy emphasis on service-oriented architecture, and quality and automation. The tradeoff between the tightly coupled stages of Waterfall and the parallel and fast-moving nature of Agile is building things independently, and test-driven development and automated deployment; recognizing that you can't easily ensure quality on features delivered out of sync with human intervention, the best method for ensuring quality without hampering efficiency is to bake quality into the development process. Unit tests before engineering, unit and integration test coverage at maximum, staged deployments with auto-rollback, 1-box or n-box releases; all are designed to make software development fast and production releases as low-touch and low-risk as possible.

(Slide 5)

Scrum is the most basic form of Agile. In traditional scrum, you get execution teams that will operate on narrow, time-boxed work, called tasks, which are bundled into stories representing deliverables, which are bundled into epics, which represent projects. Everything in Scrum is driven by the product owner. This person is responsible for coordinating between customer, business, and development. This person will be developing the backlog of stories and managing epics, sitting in on and guiding planning and retrospectives, and performing acceptance testing. Scrum works fast, but still does have a concept of work buckets or deliverables; we call them sprints. Best at 1 or 2 weeks, they can sometimes stretch to 4, and have a predictable cycle. The product owner builds a backlog of stories and prioritizes them, then the team works with a scrum master to put together an acceptable bucket of work for the sprint in planning. With a sprint agreed and started, daily scrums keep the scrum master up to date on progress and can inform the product owner of issues, and at the end of the sprint a retrospective is performed. This process is pretty well known, and most teams play a little with the edges of the framework, but it can be quite effective.

Which brings us to: Lean development!

(Slide 6)

Lean engineering-- brought to you by Toyota! --is, I would say, one of two natural evolutions of scrum; the drive towards absolute efficiency, the just-in-time manufacturing techniques brought to programming. Lean argues that with work parallelized, teams working independently, and quality now baked into the process, the focus of engineering should be the elimination of all waste, stretching the meaning of late decisions and fast deliverables as far as they will go, and allowing developers to work mostly without restraint from outside influences. Lean flattens companies, empowers engineering to make decisions and set requirements and talk directly to customers (cut out the product middleman), sets the team to continuous deployment, and allows work to be truly continuous. Lean takes many functional forms, but the most popular, which we will talk about now, and my favorite, is **Kanban**.

(Slide 7)

What is Kanban?

Kanban is all about taking the principles of Scrum, and applying Lean philosophies, stripping away everything non-essential and allowing work to flow naturally. When we look at a standard sprint, what can we identify as non-essential? Well, for starters, the product owner is mostly a middle man in Scrum, connecting people and trying to flesh out details. That role is better suited when focused on the backlog and not worrying about how the team delivers. We also see that the concept of a sprint isn't really needed; if we have a robust and ever-growing backlog of work, why do we need formal planning? Or a formal retrospective? Prioritize stories, and queue them up; this also allows us to not worry about assignment. Team members don't need to be given work, they just take whatever is up next in the queue. We move task tracking into multiple queues; backlog, in progress, in review, and done (with a special blocked queue to handle issues). This gives a natural flow to work, without overwhelming the engineer or always panicking the team about end of sprint deliverables. You can still monitor and track velocity, but on an ongoing basis.

Which brings us to: Extreme!

(Slide 8)

This sounds way more fun than it really is, but Extreme Programming is really just the logical end-stage of multiple trains of thought in any development framework. For example, if we have code reviews as a means of enforcing quality, then Extreme says, “can we make code reviews robust and continuous? And if we make them continuous, why not just put the reviewer side by side with the engineer?” And thus we birth: pair programming. Extreme also says, "if we have two-week deployment cycles, why can't that be 1-week? Or 3 days? Or 1 day? Or 1 hour? Or immediately?" And thus we birth: continuous integration. CI/CD means that we have very tiny releases with no formality, which means we don't really need big release programs; we can flatten the whole development team and make releases simple. Extreme then says, "since requirements can change, and we’re not flat, why don't we just gather requirements when we're ready to use them?" We remove standard planning and require frequent communication with stakeholders. We take the sample principle to testing, and so on…

It's all about going to the Extreme conclusion.

"Assume X; if we extend that as far as we can, what does X become?" It's not really so extreme, honestly-- we're not skydiving here --and it has a lot of drawbacks; paired programming, for example, can be resource wasteful; a lack of planning can muddle requirements and expectations, automated acceptance testing can be prone to human error that might be otherwise catchable, etc. But it is a good thought exercise when developing your own framework.

(Slide 9)

A key to all this effort is finding a way to deliver work to the customer as fast as possible and in as good a quality as possible. Test-drive development is core to this mission; you can't dev fast and deploy fast unless you can be reasonably certain that you will catch most basic failures before getting to the customer. Unit tests should be built before code based on requirements, integration tests should extend coverage to all workflows, which has very clear environment demands, and pipelines should be built to execute those tests and manage deployments with minimal human involvement (or none).

(Slide 10)

Software to handle this work should be fairly extensive, able to grab code from git or svn (but please use git), build (if necessary; scripting languages likely only need some configuration management), and scripting to handle deployment staging and other custom elements unique to each system. You want a system that can execute unit tests, wait for integration tests to complete and environments to reset, scan and understand logs during each stage of release, deploy to n-box and detect failures, then deploy to all production or roll-back and alarm.

There are a lot of good options out there-- Jenkins, Continuum, Shippable -- there's dozens, really, but my favorite to date has been Jenkins. It's light weight, extensible, easy to work with, easily scriptable, and can handle all the major pieces of the puzzle almost out of the box.

Oh, and it's free!

(Slide 11)

As for managing your day-to-day work (Scrum or Kanban), the field has gotten very crowded. You want something that will effortlessly track stories, tasks, and defects, and tie back to git. You want software that can provide good monitoring and reporting, track the burndown rate and ongoing team velocity, as well as adapt to include documentation. And you need these tools to be accessible, which means web-based; don't bother with anything tied to the desktop.

Oh, I should also quickly explain burndown and velocity:

Burndown charts are graphical depictions of the work remaining on a given timeline. In Scrum, the timeline is almost always the current sprint; in Kanban, it's variable based on the business requirement. Velocity is related, and is simply the amount of work completed in any given unit of time prior; if tracking the trailing 2 week velocity, you take 2-week chunks prior to the current week and tally the completed story points per developer. A manager will track trailing velocity over time to look for dips or rises in productivity.

(Slide 12)

Anyway, the original standard software system for managing developers was *Basecamp*, written in 1999 by 37signals. It was primarily a bug-tracking and project management tool. This isn't really what you want, although they are very, very good at exactly what they do, and the software stack has expanded a lot since the last time I used it, so my knowledge may be out of date (don't take anything I say here as gospel). Then there is *trac*, released in 2006, which was mostly also a bug-tracking tool but was made to be highly extensible. Those who use trac typically create, essentially, their own tooling after a short while of adding and modifying plugins and customizing steps to suit their individual need. I would say most teams don't really want trac, either, though...

(Slide 13)

Then we have a few upstarts, as I like to call them: Great web-based tools that are awesome at what they do, but not quite up to the level of pluggability and extensibility that I usually want. Rally, for example, is a great web-based tool that lasers-in on Scrum, offering great tooling for executing a standard sprint. Trello, meanwhile, lasers-in on Kanban, offering an awesome queue-based tracking system. And Trello even has a free option, and is probably the tool I would recommend to a start-up that is cash-strapped and looking to operationalize.

(Slide 14)

But the standard, I have to say, is *JIRA*. It's been around a long time, originally released in 2002, and now is part of a huge suite of services offered by Atlassian, such as Confluence, Stash, and HipChat. JIRA is powerful, pluggable, has strongly baked extensions for Scrum and Kanban (and other techniques also), has a very robust reporting and monitoring suite, integrates with all kinds of source control, has active alerting, manages users, has a good code review framework, and can be integrated with CD/CI tooling. And it's generally affordable for most companies with at least a little cash; small groups can be $10/month or less. I will always recommend JIRA, and other Atlassian products, for a growing company looking to move fast and institute a robust Agile process. I know I sound salesman-y, but I promise I have no connection with Atlassian; I just like their stuff.

(Slide 15)

So, whatever you do, always remember: making software is hard. Don't let your tools and process make it any harder. Nothing is gospel, each organization should build a development process that fits with their culture and product. I, personally, love working in Kanban; it's fast, smooth, and generally quite productive, and it is very easy to track without feeling like a micromanager. But that’s only my preference; use whatever works for you, and just make sure it's fast.

Move fast and break things, that's how you make the best stuff.