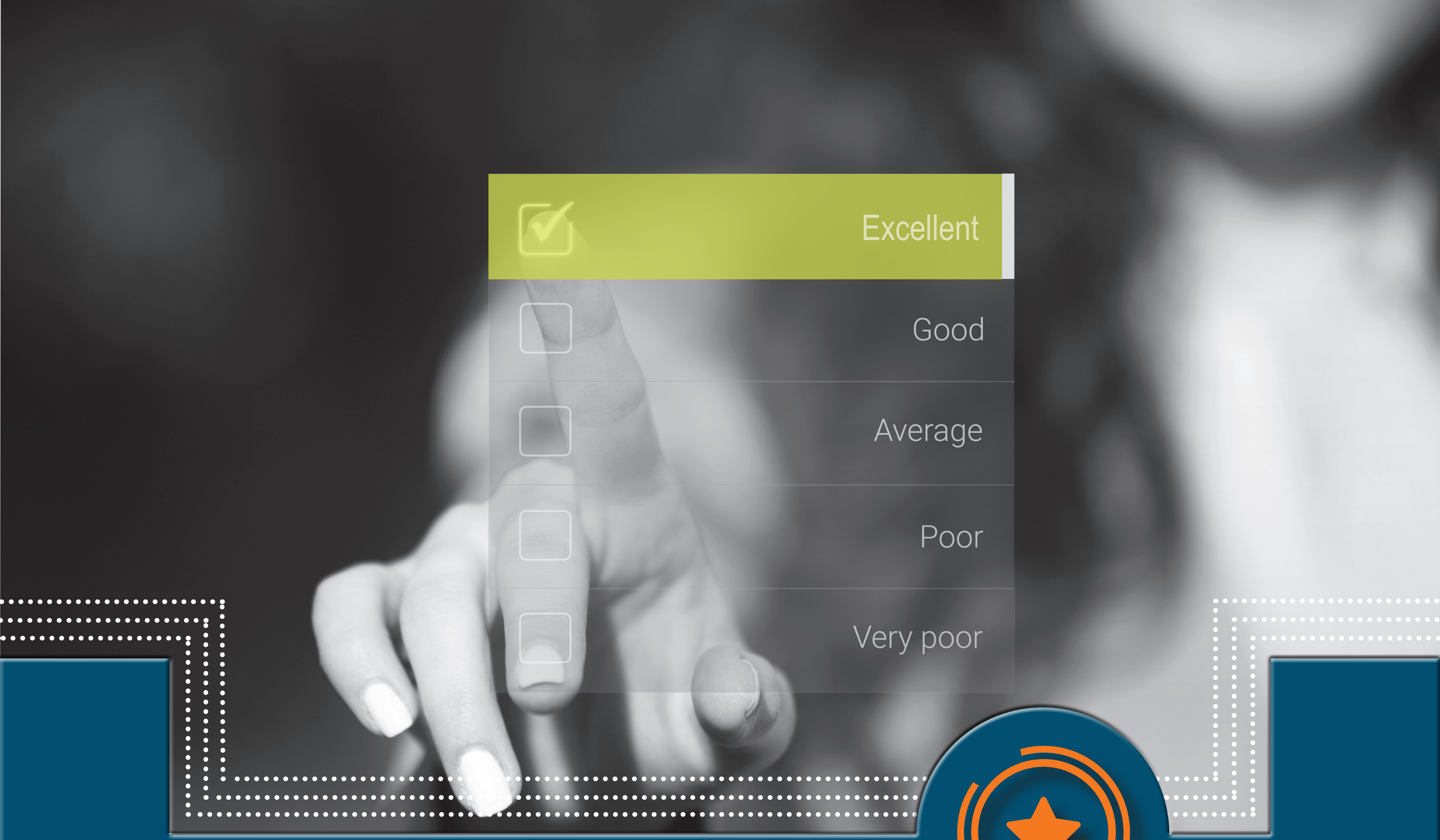


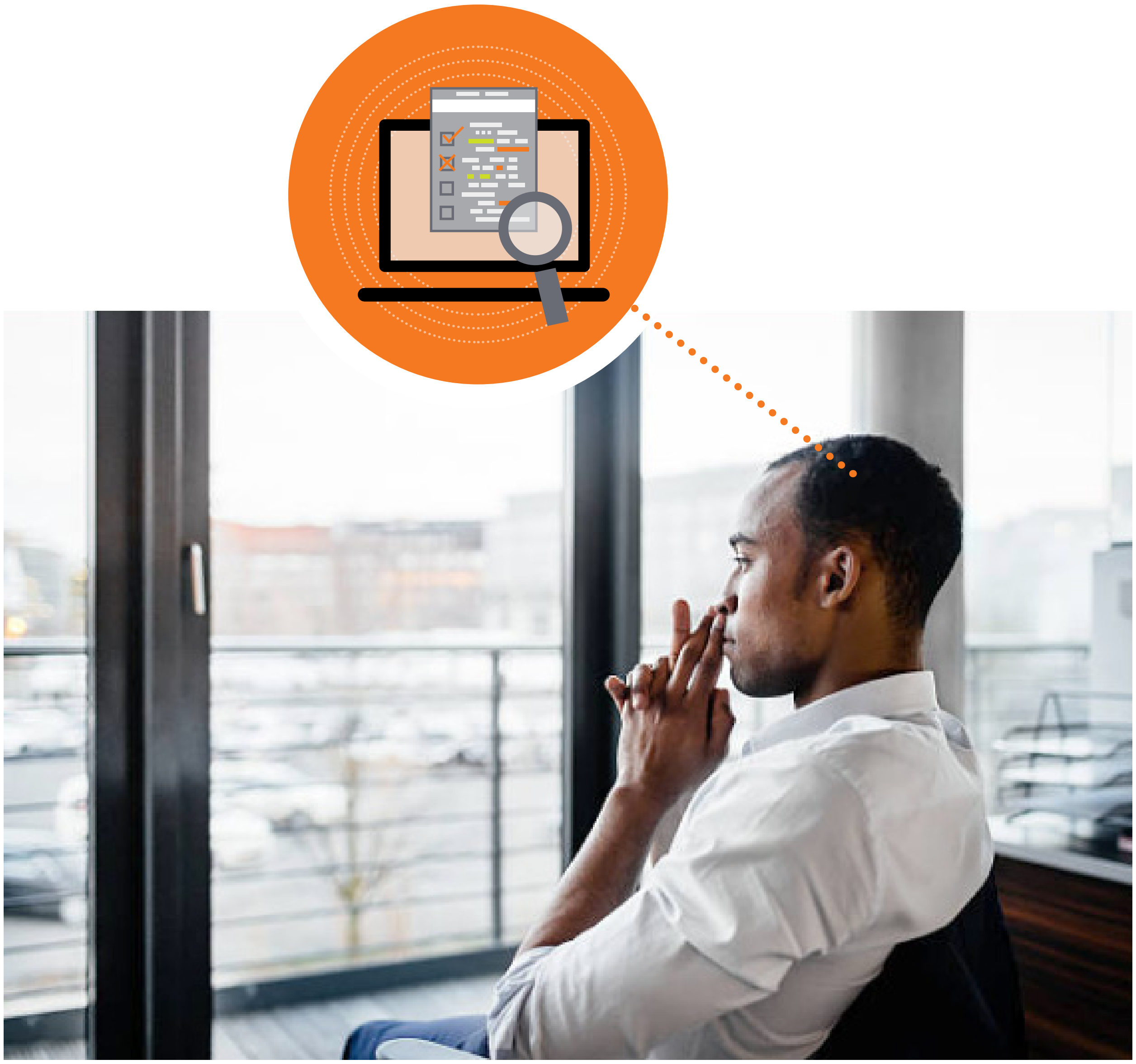
How much quality do you want?

Improving and measuring quality in application development

By Jared Darling
Solutions Delivery
Application Development

aim consulting
AN ADDISON GROUP COMPANY

“How much quality do you want?” This is a question that is never asked in application development, but it should be—up front and often.



Application development teams commonly view quality as a step that happens at the end of the development process and the responsibility of the quality assurance team (QA). At first glance, this seems reasonable as it is the job of QA to ensure that code passes various tests, that the application is stable and trustworthy, and that new features meet user acceptance criteria.

But in reality, quality is not determined by whether or not a box is checked. Quality starts at the beginning, when goals and outcomes are defined, a timeline is established, and resources are deployed, and continues throughout the entire application development process, from architecture and design straight through to deployment and round again with every new feature release.

Even if everyone understands this conceptually, when quality suffers, most teams try to improve it by adding tools or "best practices" like agile, automation or DevOps. The problem is that a lot of teams have a tendency to jump to solutions before understanding the problem.

When you approach software quality analytically and comprehensively, you can ensure that not only are quality standards met, but will be able to answer the questions "how much quality do we want?" and "how much quality did we provide?"

When you approach software quality analytically and comprehensively, you will be able to answer the question *"how much quality do we want?"*

Relying only on Tools and Best Practices is Ineffective

The main causes of software quality issues are understood to be one of the following:

1) Schedule slippage forces the QA team to rush with deadlines being missed

2) There are too many bugs, resulting in time wasted in rework

3) Business requirements are not being met.

Teams often respond to these issues by reviewing the newest best practice(s) to increase quality in software development and implementing the supporting tools or practices into their processes. Automation is one example. Agile is another. The hottest ticket currently is CI/CD or DevOps, which is sort of an extension of both agile and automation together.

This is not the most effective way to go about it.

It's not that automation, agile, or DevOps are bad ideas. Alone or together, implementing modern best practices for software development can result in better application quality if they're understood well and implemented correctly. The problem is that many teams merely install tools that support these practices, only to be disappointed in the results.

It also depends on what the quality issues actually are.

For example, consider an example scenario for number **3) business requirements are not being met**.

There may be many reasons for a new feature failing to meet business requirements, but a common one is that the intent of a business need got lost in translation when articulated as a user story and its corresponding acceptance criteria.

Here's how technical teams might try to solve this problem:

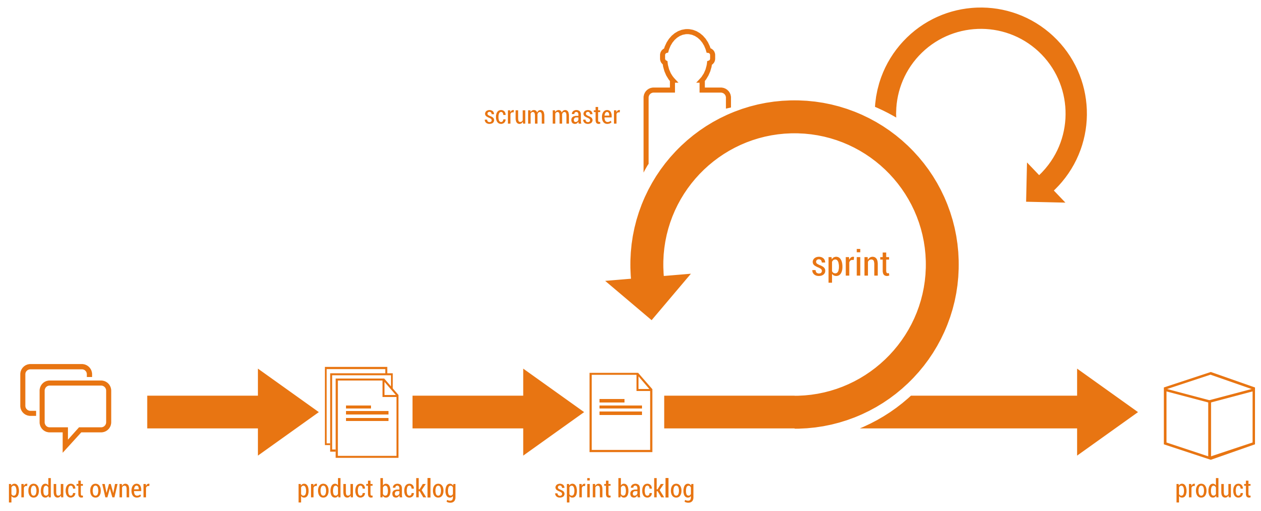
Before doing anything else, they turn to tools. For example, they might decide to use Cucumber, a tool designed to aid in the automation of acceptance tests by providing a way to express test cases in natural language. They adopt the tool hoping it will "improve quality" but, failing to understand its intent, they use it as a repository to dump technical requirements for automation. Of course, quality doesn't improve at all and the team becomes frustrated, feeling that Cucumber is a useless tool that just adds an unnecessary step, when in reality they didn't scrutinize the real issues or evolve at all. Instead of adopting the natural language that Cucumber is meant to inspire, they just added a tool to their process and continued to write user stories as technical requirements the way they had been doing before.

In other words, tools alone don't solve problems; more often than not, the issue is in your processes.

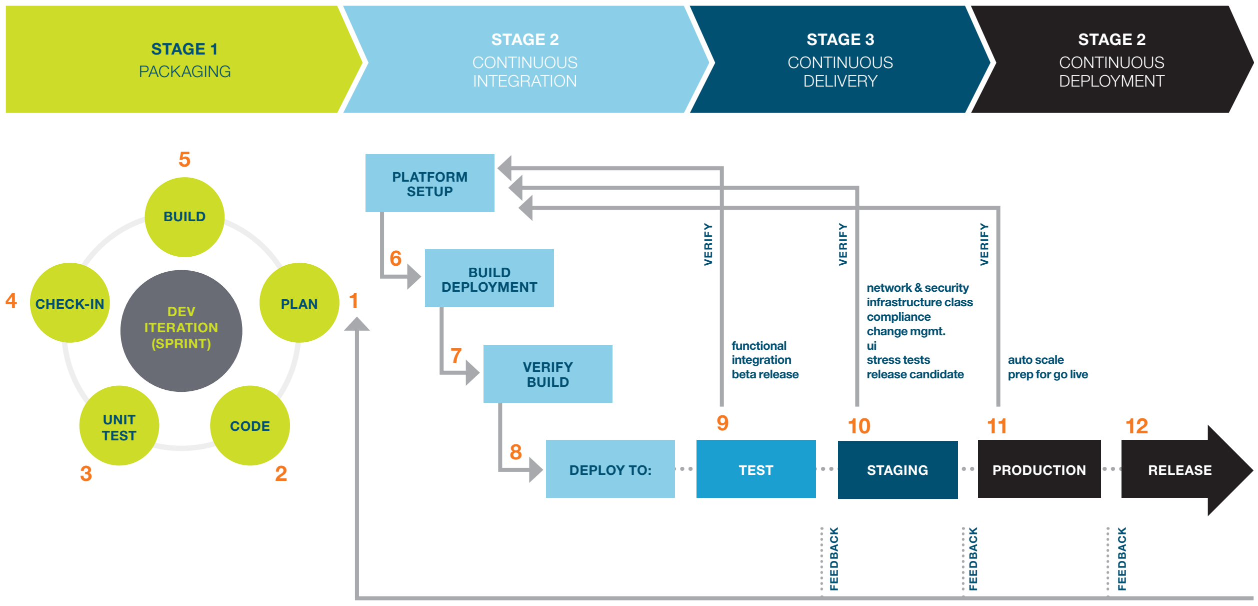
In other words, tools alone don't solve problems; more often than not, the issue is in your processes.

Or take agile development, widely acknowledged as a best practice for increasing quality. One of the more common problems with agile is not properly accounting for QA as part of the sprint cycle. With agile, development work is boxed into iterative periods of time, called "sprints" in Scrum methodology, which is supposed to end with shippable software.

The whole team comes together during planning to estimate the features that can be completed in a sprint, but it's common for developers to either underestimate the time they need to develop the features, or not include quality tasks as part of the sprint, resulting in releases of poor quality or requiring the team to put in overtime. Sometimes developers will keep developing, moving on while testers rush to complete the work. This enforces the common misconception that when it comes to application quality, developers own the code and testers own the quality. Evolving this into a more cohesive approach is essential.



CI/CD and DevOps are intended to address this issue. Very simply stated, Continuous Integration (CI) is a practice designed to automate build processes, version control, and testing. Continuous Deployment (CD) extends that practice through the deployment process. When development, QA and IT operation teams are collaborating and communicating together on the process of software delivery and infrastructure through automation of building, testing, and releasing, you have a culture known as DevOps.



The problem here is that many teams confuse Continuous Integration with the tools that support it. CI servers like Jenkins, Codeship, Hudson or Cruise Control help teams automate workflows, but out of the box they only contain capabilities; they still have to be configured by engineers. Oftentimes, a small team will set up CI using a specific tool for a limited use case. Liking the result, the business will want to expand CI to other teams without considering that different teams may have different automation needs requiring different capabilities or configurations. When businesses don't properly invest in CI, it can result in a negative perception and the conclusion that CI/CD "doesn't work" because of a particular configuration or use case that is not able to be met.

This brings us back to the question "how much quality do you want?" Because it's one thing to ask the business "do you want quality?" (the answer will always be "yes") and another to calculate the investment of time, resources and specification required to achieve comprehensive, high-level, measurable quality across the board and in every release.

It's one thing to ask the business "*do you want quality?*" and another to calculate the investment of time, resources and specification required to achieve comprehensive, high-level, measurable quality across the board and in every release.

The Effective Way to Improve Application Quality

To address the real issues, engineering teams need to view quality comprehensively rather than as a component of the development process. But this isn't merely a matter of applying "best practices" as previously discussed. It requires cultural change, beginning with individual and team accountability. Secondly, it requires sober analysis of the actual issues at play and understanding of the real investment to address those issues. Only then, when you know what you actually need, are best practices going to get you what you want.

Finally, more mature organizations are able to measure their results, answering the question "*how much quality did we provide?*" Here's how it works:

1. Establish accountability

Application development is both an individual and team effort. Developers need to take responsibility for the quality of the code they push to production, but testers need to step up too. It's common in sprint planning meetings for QA to take a backseat and assume a supportive role. QA needs to be vocal about what they need to do their job correctly, which may mean being more assertive during sprint planning to ensure QA has enough time, especially if they are getting crunched repeatedly in every sprint. Indeed, QA needs to assume authority for an application's quality end-to-end, and not merely as an end-of-the-process checkpoint. Is it testable? Does the application meet business requirements? Is risk properly managed? Over the long haul, this strengthens communication, mutual respect, and capability within development teams.

Everybody involved in the application should be equally invested in the culture and processes that lead to quality applications.

Also, if DevOps is your destination, wrap in the IT/Ops team so that everybody involved in the application—from infrastructure to development to testing to release—is equally invested in the culture and processes that lead to quality applications.

2. Understand the real problems

Sleuthing out the source of quality issues is needed, especially when those issues are chronic, with questions asked and answered as a team together and with honesty:

- **What exactly is happening?** Is the code that gets checked in overly buggy? Are cycles taking too long or deadlines being missed? Is functionality not meeting expectations?

- **Why is it happening?** Are the tools being used the right ones and configured in the right way? Are there poor design patterns in effect, resulting in a need for a lot of tricky customization and rework and expanding technical debt? Are roles and processes crystal clear to everyone on the team? Is the team adequately protected and able to focus on their commitments? Are user stories and their acceptance criteria fully understood?

Identifying the source of quality problems is important as it is not uncommon to see teams employing a process solution for a technical problem or a technical solution for a process problem.

3. Set clear goals

To maximize efficiency and ensure the right solution is applied to the problem, articulate clear goals such as “eliminate high priority defects.” If possible, depending on the specifics of the project and team, take goal setting a step further with measurable KPIs such as “we want to reduce sprint cycles from four weeks to two” or “we want to decrease the number of defects per release by x%”. Setting measurable KPIs clarifies which options are the best ones, results in better planning, and will get you closer to the results you ultimately want.

Next, document the plan needed to achieve these goals, including what is needed for your internal teams to function better or produce higher-quality applications. One useful form of documentation is the Definition of Done, or working agreement, used to assess when a user story is completed. Using this model helps to identify any lack in quality—and in the vast majority of cases a team will find that the problem is in the process, not the tools.

Once you have goals, you can start to answer the question “how much quality do you want?” In other words, what is the right level of investment to achieve the level of quality the business really needs?

Not all applications are the same. Quality assurance encompasses both risk and complexity and it’s important to acknowledge that the standard to be met varies. For example, a certain level of risk can be tolerated for an e-commerce application while an application powering a rocket ship sending precious cargo into outer space must be perfect—in other words, buggy software is a bigger deal in some applications than others. As intolerance for risk increases, so should the level of investment in quality assurance. This is rarely a question businesses grapple with, but can be tackled in practical terms.

Depending on the quality issue you are solving, as well as how much defects matter to your application, there are varying levels of investments that can be made to improve quality—from adding some simple automation to rebuilding the application from the ground up with better design patterns, armies of test engineers, and a full-blown DevOps culture.

4. Implement best practices

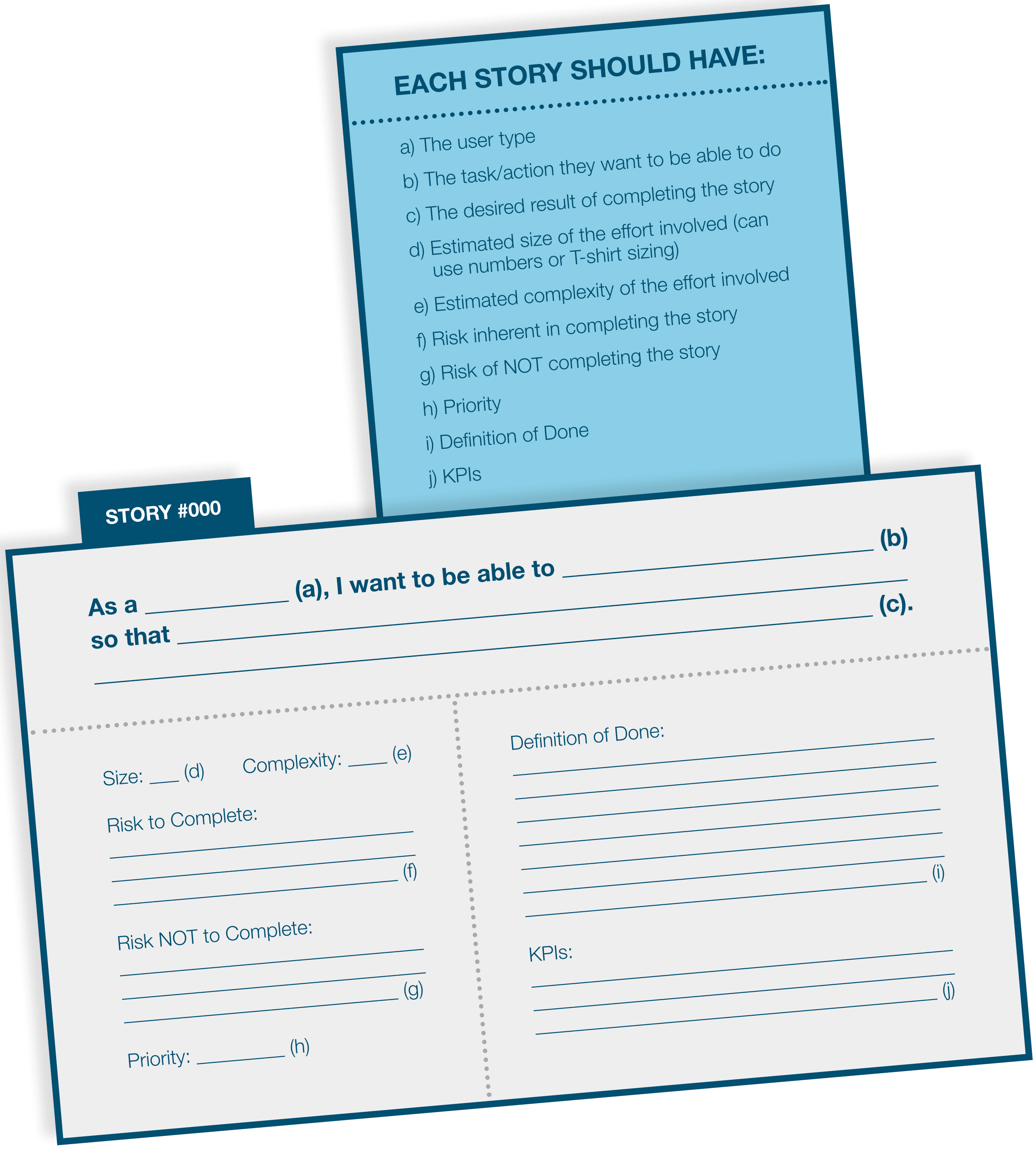
This is the step that a lot of teams jump to without first doing their due diligence. Now that the team owns accountability for quality, the real problems have been identified, and the objectives are clear, it's time to look at which best practices and tools are best suited to get you from A to B.

Through all of the previous steps, the team should be encouraged to remain as abstract and agnostic as possible when discussing any possible solutions to the quality problems being addressed. That discipline will pay off once the team reaches this step.

With the amount of effort put into the previous steps, the gaps in quality should be very clear. At this point you might want to leap into assigning solutions to problems, but it is recommended to take thoughtful, deliberate action to obtain a measurable result.

A strong approach recommended by AIM Consulting is to articulate quality gaps as user stories, *the same way you would regular work*. In other words, create a prioritized backlog for addressing quality issues in the software development process itself.

The first three elements make up a basic user story—who wants what and why. Size, complexity and risk are essential for determining the level of investment and how much quality you want. This could also be displayed as story points. Documenting the risk of NOT completing a story is important for prioritization, especially if the size and complexity of the user story is also high. Finally, the Definition of Done and the associated KPIs (when viable) are going to allow you to measure quality.



EACH STORY SHOULD HAVE:

a) The user type
b) The task/action they want to be able to do
c) The desired result of completing the story
d) Estimated size of the effort involved (can use numbers or T-shirt sizing)
e) Estimated complexity of the effort involved
f) Risk inherent in completing the story
g) Risk of NOT completing the story
h) Priority
i) Definition of Done
j) KPIs

STORY #000

As a ____________ (a), I want to be able to ____________ (b)
so that ____________ (c).

Size: ___ (d) Complexity: ___ (e)

Risk to Complete:
____________ (f)

Risk NOT to Complete:
____________ (g)

Priority: ________ (h)

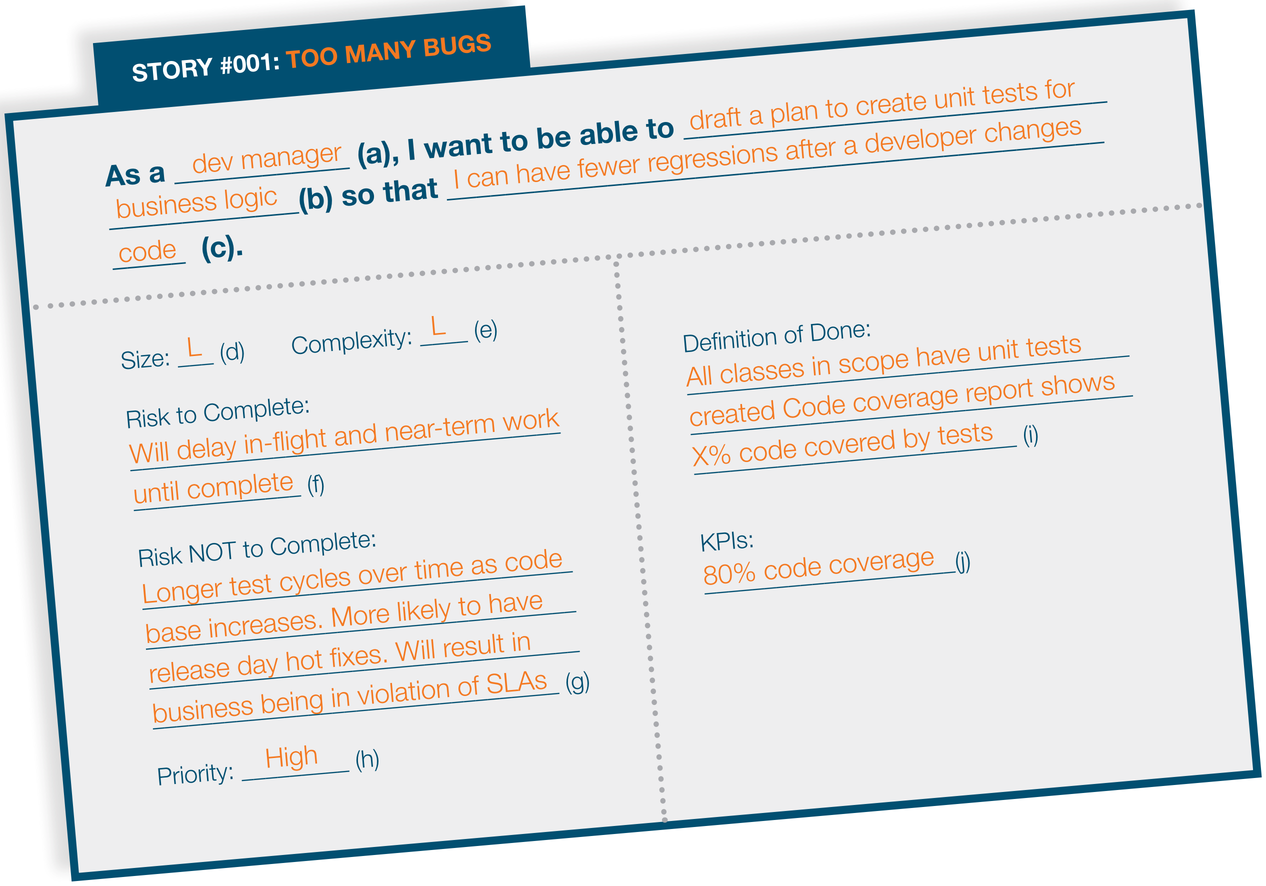
Definition of Done:
____________ (i)

KPIs:
____________ (j)

Let's look at some examples

1) There are too many bugs

If the problem is that "code is really buggy", dev managers (user type) might decide they want to draft a plan to create unit tests for business logic (action), to have fewer regressions after a developer changes code (desired result). The size and complexity of this story is large. The risk of doing unit tests is that it will delay in-flight and near-term work until the tests are complete. However, the risk of not doing unit tests is suffering longer test cycles over time as code base increases with a greater chance of needing hot fixes on release day and being in violation of SLAs. This makes the priority high. Completion would mean that 'all classes in scope have unit tests created' (Definition of Done) with the percentage of code coverage tracked. To meet a specific KPI, the team might conclude to implement a popular tool such as SonarQube for code coverage and analysis, and assign a coverage completion target metric of 80%.



STORY #001: TOO MANY BUGS

As a dev manager **(a), I want to be able to** draft a plan to create unit tests for business logic **(b) so that** I can have fewer regressions after a developer changes code **(c).**

Size: L (d) Complexity: L (e)

Risk to Complete:
Will delay in-flight and near-term work until complete (f)

Risk NOT to Complete:
Longer test cycles over time as code base increases. More likely to have release day hot fixes. Will result in business being in violation of SLAs (g)

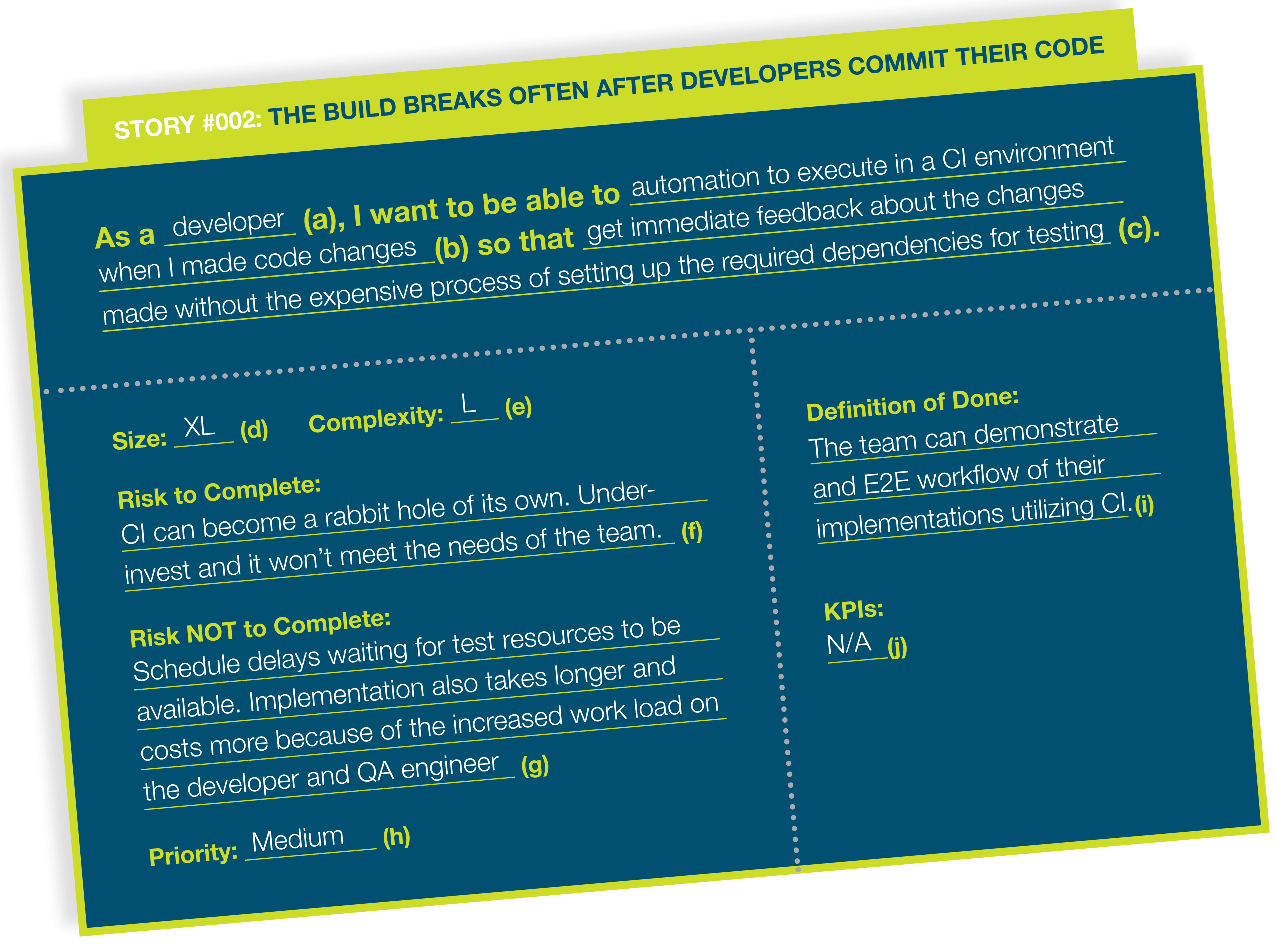
Priority: High (h)

Definition of Done:
All classes in scope have unit tests created Code coverage report shows X% code covered by tests (i)

KPIs:
80% code coverage (j)

2) The build breaks often after developers commit their code

If the quality problem has more to do with code breaking after it is submitted, then the developer (user type) might want to implement automation and continuous integration (action) to get immediate feedback about the changes made without the expensive process of setting up the required dependencies for testing (desired result). However, CI is an entire paradigm shift requiring significant investment so this should be noted as an XL effort with considerable complexity. There's also the risk that CI can become a rabbit hole. Under-invest and it won't meet the needs of the team. However, the risk of NOT implementing CI includes schedule delays waiting for test resources to be available. Implementation takes longer/costs more because of increased work load on developer and QA engineer. Prioritization will depend on the risk of not implementing CI and the level of investment the organization is willing to make, but in this example, it's probably a medium. A Definition of Done might be "the team can demonstrate and end-to-end workflow of their implementations".



STORY #002: THE BUILD BREAKS OFTEN AFTER DEVELOPERS COMMIT THEIR CODE

As a developer **(a), I want to be able to** automation to execute in a CI environment when I made code changes **(b) so that** get immediate feedback about the changes made without the expensive process of setting up the required dependencies for testing **(c).**

Size: XL **(d)** **Complexity:** L **(e)**

Risk to Complete:
CI can become a rabbit hole of its own. Under-invest and it won't meet the needs of the team. **(f)**

Risk NOT to Complete:
Schedule delays waiting for test resources to be available. Implementation also takes longer and costs more because of the increased work load on the developer and QA engineer **(g)**

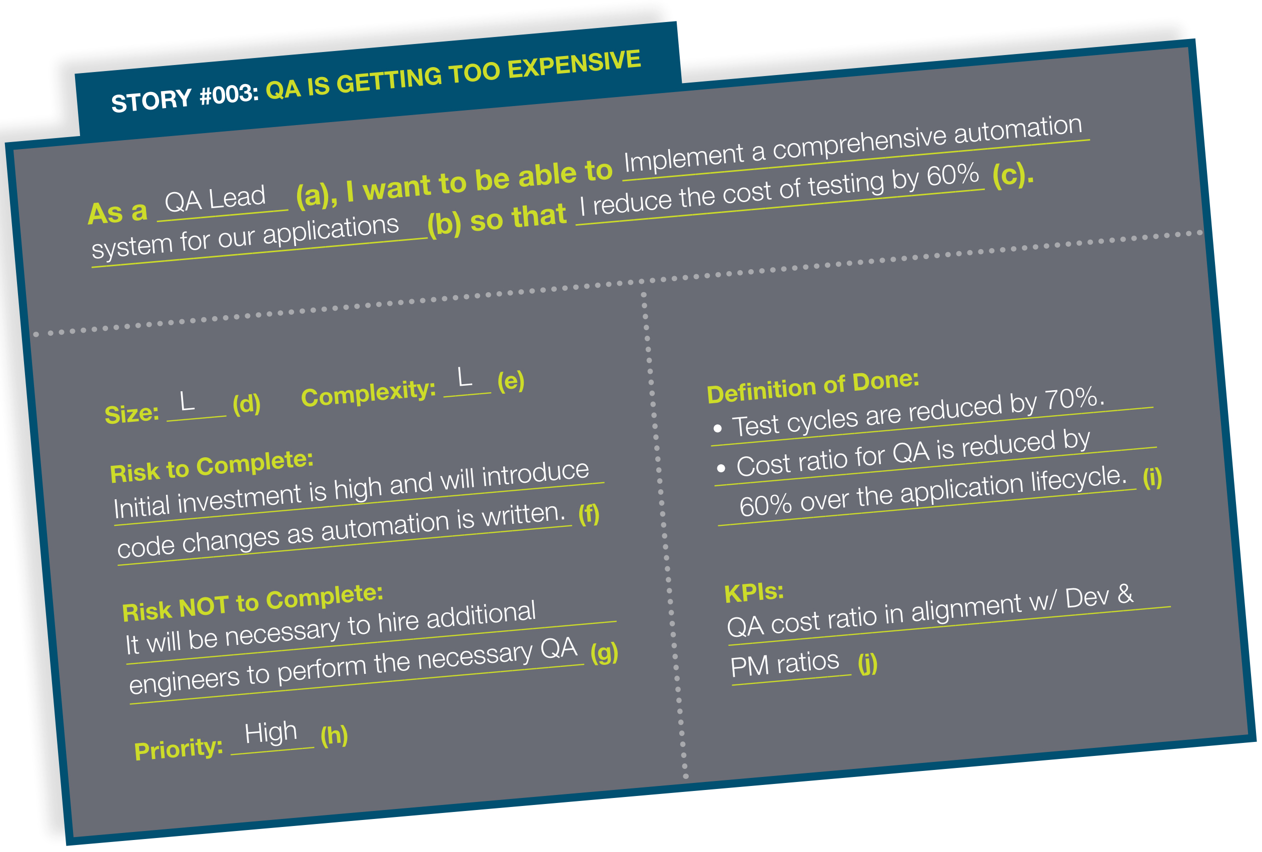
Priority: Medium **(h)**

Definition of Done:
The team can demonstrate and E2E workflow of their implementations utilizing CI. **(i)**

KPIs:
N/A **(j)**

3) QA is getting too expensive

Let's assume that a business is struggling with an expensive and time consuming QA process. The QA lead (user type) might want to implement a comprehensive automation system for applications (action) in order to reduce the cost of testing by at least 60% (desired result). The size is Large. The complexity is Large. The risk is a high initial investment that will introduce code changes as automation is being written. The risk of NOT implementing automation is that it will be necessary to hire additional engineers to perform the necessary QA. This makes the priority High. A Definition of Done and associated KPIs would be the cost ratio for QA reduced by 60% over the application lifecycle with reduction in test cycles by 70%. Additionally, QA ratios should be in alignment with Dev and PM ratios.



STORY #003: QA IS GETTING TOO EXPENSIVE

As a QA Lead **(a), I want to be able to** Implement a comprehensive automation system for our applications **(b) so that** I reduce the cost of testing by 60% **(c).**

Size: L **(d)** **Complexity:** L **(e)**

Risk to Complete:
Initial investment is high and will introduce code changes as automation is written. **(f)**

Risk NOT to Complete:
It will be necessary to hire additional engineers to perform the necessary QA **(g)**

Priority: High **(h)**

Definition of Done:
- Test cycles are reduced by 70%.
- Cost ratio for QA is reduced by 60% over the application lifecycle. **(i)**

KPIs:
QA cost ratio in alignment w/ Dev & PM ratios **(j)**

Once the team has a list of stories that articulate the quality gaps and an idea of how much investment each will take, a conversation around 'how much quality do we want' can intelligently be had.

5. Measure how much quality you provided

Whatever best practices you choose to implement, you can measure overall effectiveness by measuring how much quality your teams provided with each release. Most application development teams measure quality through acceptance tests. Does the application feature meet the business requirements, yes or no? This is a fine place to start. However, you can take it a step further by applying quantitative metrics to measure quality output sprint by sprint.

Here's how it works: at the beginning of a sprint, count up the number of acceptance criteria that have test cases (this is preferably all of them). At the end of a sprint, determine the number of criteria that were met. Divide the result by the goal and you have a percentage metric that can be tracked over time, sprint to sprint. Like velocity, this metric can be used to evaluate the performance of your team. If your test cases are automated, this is extremely easy.

The KPIs identified in the previous step can also be utilized to provide actionable information that the team can use to more efficiently target quality issues when they next come up. Here's where analytics can play a role in automation. Automation logs can easily be harvested from CI servers and stored in databases for processing. Results for automation test cases over time will begin to emerge. Utilizing BI analytics tools, directors of engineering and business stakeholders can have visual dashboards that present the quality of their applications in near real-time.

Final Thoughts

One important factor to understand is that the desire to improve quality really only occurs at the point in which the team or business decides that problems with the application are encroaching on the ability of the software to provide value. This can happen anytime during a project—while creating the project backlog (planning phase), during defect resolution (delivery phase), or even after the project has completed (project retrospective).

Regardless of the timing or reason, development teams need to adopt a comprehensive view of quality and approach new tools and best practices with a critical lens for solving their specific issues. Too often, technical people want to jump to technical solutions, essentially skipping steps 1-3 (accountability, assessment, goals) in favor of 4 (implement best practices and tools). To really achieve quality, the commitment needs to be shared across team members, the goals need to be clearly defined before a plan is made, and the progress needs to be measurable.

Ultimately, application quality matters to everyone. Evolving a culture requires transparency that allows viewing of quality from both the organizational and individual level. This will evolve the development process and its individual components that not only builds morale across the team but strengthens the business as a whole.

We build outstanding software

Our approach to application development is team-centric and holistic, from architecture and design, to modernizing software development processes, to providing flexibility to our clients to engage our experts through managed services to clear a stacked backlog, resolve issues, and scale for long term strategies.

Software Strategy and Design
- Assessments and strategies
- Planning and roadmaps
- Enterprise application architecture

Custom Web Applications
- Full stack web applications
- Service oriented architecture
- API development
- Cloud platforms
- QA and automated testing
- Deployment and maintenance

Development Best Practices
- Agile Scrum, Lean-Agile, Kanban
- Scaled and SAFe Agile
- TDD and Paired Programming
- CI/CD and DevOps

About AIM Consulting

AIM Consulting, an Addison Group company, is a rapidly growing leader in technology consulting services and solutions delivery that helps companies compete effectively in the digital world. AIM builds long-term relationships with the best technology consulting talent in the region and delivers end-to-end on business-critical initiatives with modern technologies and processes. Learn more at **aimconsulting.com**.

CONTACT US

aim consulting

Denver
720.305.9818
contactdenver@aimconsulting.com

Minneapolis
952.314.7255
contactminneapolis@aimconsulting.com

Seattle
206.624.5333
contactseattle@aimconsulting.com