Quality Engineer Learning Roadmap

A beginner's guide to the skills, tools, and technologies you need for a career as a Quality Engineer or SDET

What should you learn to become a Quality Engineer? What languages should you pick up, what tools should you master, what skills should you practice? If someone was interested in this career, where would you tell them to start—what's critical, what's nice-to-have, and what is yesterday's technology that is no longer relevant?

This isn't an easy question to answer — quality engineers depend on everything from basic computer science and quality assurance fundamentals to the latest automation frameworks, application stacks, and testing tools. Just listing everything out is daunting! In this post we will walk through a learning roadmap that describes all the skills, tools, and technologies necessary for success as a quality engineer.

The Role of Quality Engineer

First of all, what is a "quality engineer"?

While there are many ways to build software, we believe it is best created by small, cross-functional, collaborative teams. The 'cross-functional' characteristic describes a team has all the roles and skills necessary to build the solution, without reliance on external teams or skillsets.

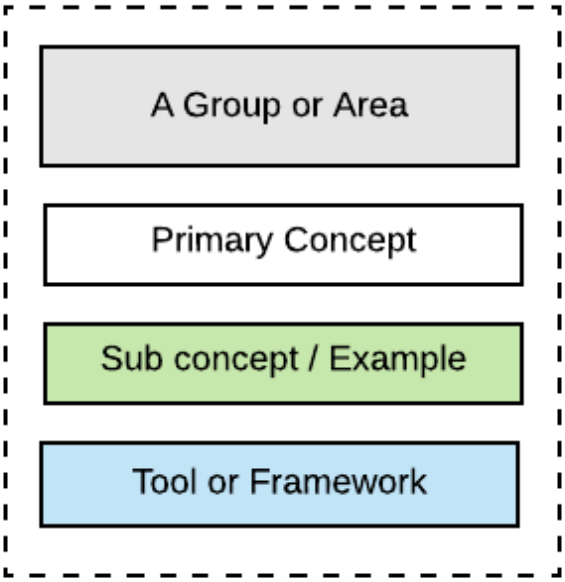
One role within these teams is the quality engineer (QE). QEs are more than just testers or automators, they empower teams by bringing a quality mindset to every aspect of building software. They are experts in quality assurance, test automation, risk analysis, agile processes, CI/CD, and everything else that can impact product quality. They collaborate with all other roles to ensure quality is built-in from day one, from the first story, before the first line of code is written. Some companies call this role an SDET (Software Development Engineer in Test), but every company defines roles differently, so what an SDET or QE does at one company might not exactly match the next.

While this roadmap was constructed specifically for our quality engineer role, it will be relevant to anyone looking to start a career in a quality related field, regardless of the name or title.

Quality Engineering Learning Roadmap

While we originally sought to keep our Quality Engineer Roadmap simple, the number of topics and skills was so large that we decided to first organize it into general areas of study. As you can see in the diagram below, there are hundreds of individual topics, all organized into a more manageable set of eighteen sections. Don't get too caught up with the full view now, as we'll walk through each section individually.

As you can see, there's a lot to learn. The quality engineer role is broad, and learning all the relevant skills is a significant investment!

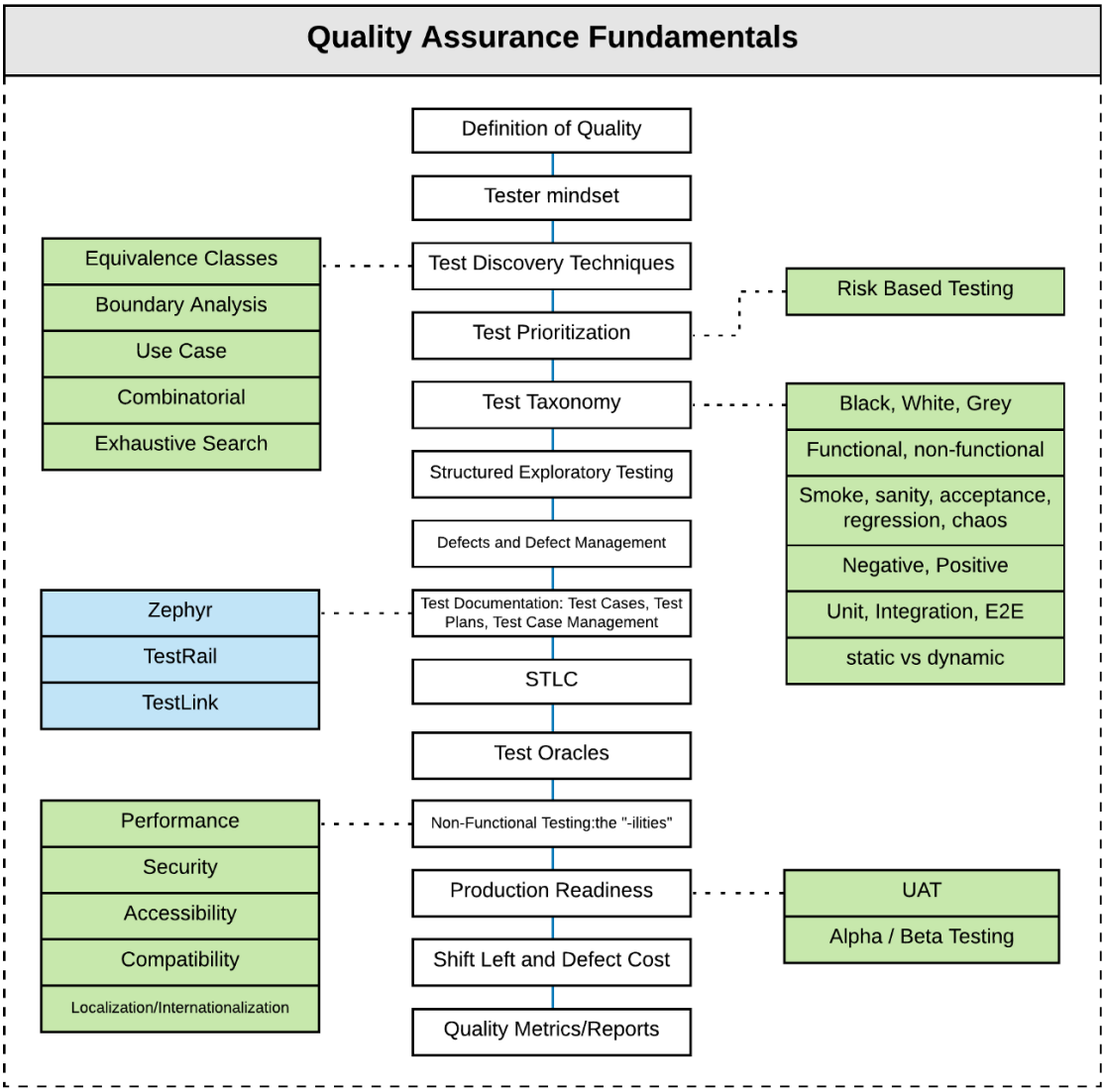


As we walk through each area, we'll use this simplified legend. Primary concepts will be in white, sub concepts in green, and tools, languages, or frameworks in blue. When you see green or blue boxes attached to white boxes, you should interpret the connecting line as "for example." The list of sub concepts or tools will not be exhaustive, but just indicative of type of things we're describing. For blue boxes, the examples given will represent what we feel are the most relevant or most valuable to the aspiring quality engineer.

So, without further introduction, on to the roadmap!

Quality Assurance Fundamentals

We start our learning journey with Quality Assurance Fundamentals. Having a strong foundation in quality assurance is critical for success, and almost every other topic builds on what you'll learn here.



To start, we need to understand what we mean by “quality” with respect to software. Knowing that, we can go on to define what a test case is and how to develop test suites and test plans using test discovery techniques. Within discovery techniques are concepts like edge cases, equivalence classes, boundary analysis, combinatorial analysis, and risk-based testing. There is specific terminology used to describe tests, so knowing how to describe, categorize, and group test cases or test types using distinctions like black box, white box, or grey box testing, functional vs. non-functional testing, negative vs. positive testing, or static vs. dynamic testing, will be important.

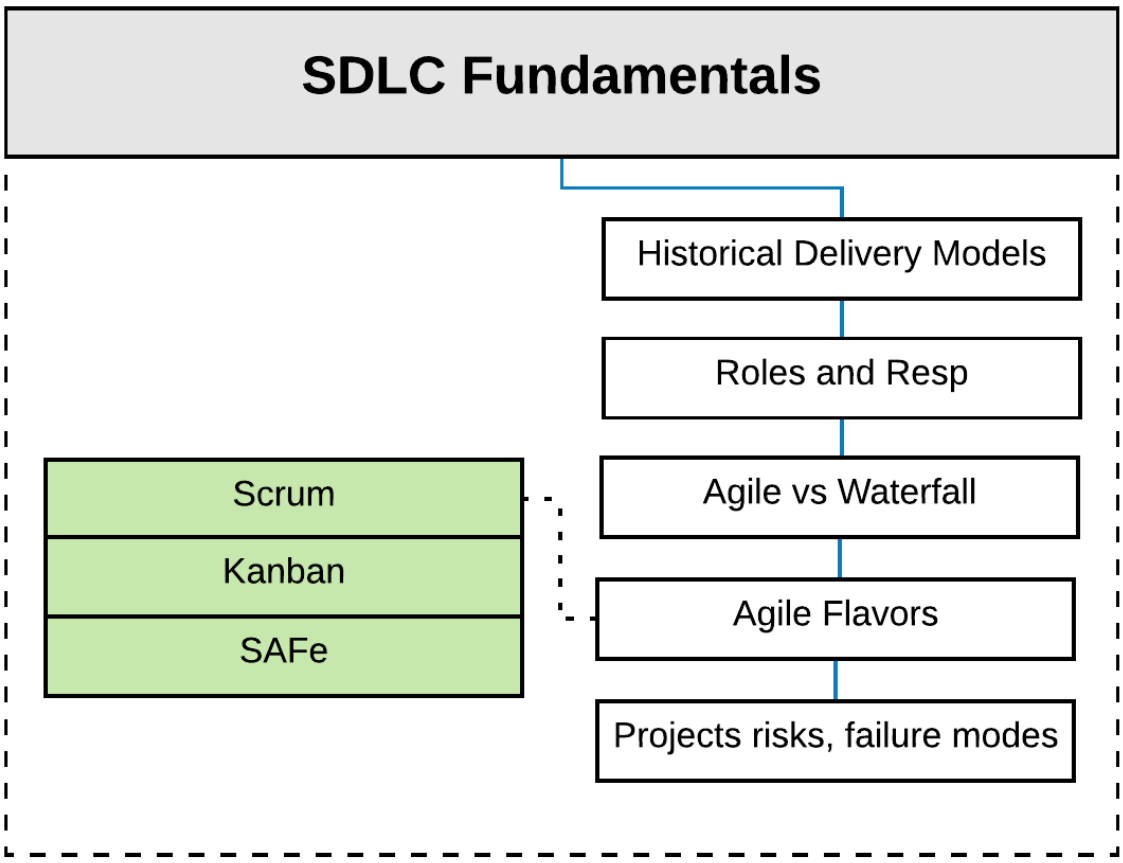
There are also activities described by non-functional testing — things like performance, security, accessibility, compatibility, localization, etc. While you do NOT need to be an expert in any of them, nor will a new quality engineer be expected to be, you should know the terms and the type of testing they describe.

On top of all this, there are tools for project management, defect tracking, test case management, and reporting to be familiar with. Jira is the most common for general project management, and tools like Zephyr, TestLink, and TestRail are common for case management.

There are more topics in this section than we have space to describe here. However, every one of them is important — it is a mastery of the fundamentals that creates great quality engineer, and a strong command of these topics is critical as you progress into later and more engineering-focused topics.

Software Development Lifecycle (SDLC) Fundamentals

Software development is no longer a solo activity, and with rare exceptions (git? Minecraft?) all modern software is built by teams. In order to be an effective quality engineer, it's important to understand how the many individuals of a software team work together to develop and ship a product. This is a broad brush category — we won't go deep on any particular methodology or development philosophy (that will come later), but we need to understand what's out there, what has been previously used, and what is currently in use today.

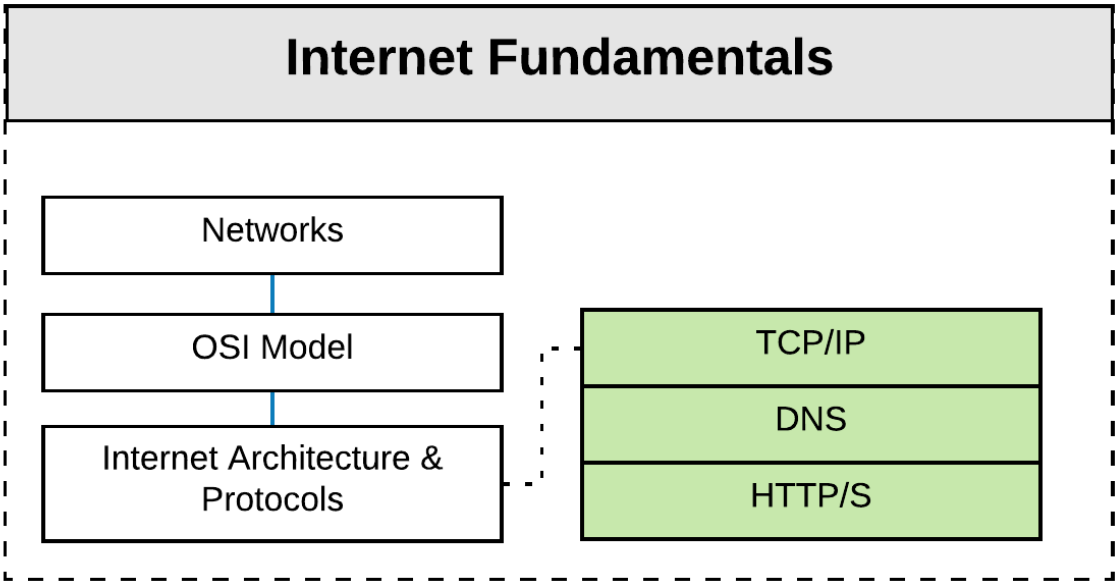


Here, you should learn what people mean when they talk about the waterfall model, how things like the V-model or spiral model predated uses of the term agile, even if they describe something very similar. You should learn the basics of modern approaches like scrum and kanban, and arguments both for and against using these types of methodologies in modern software development.

While this category might seem overly theoretical, having a strong foundation in SDLC Fundamentals will pay dividends as you begin your quality engineering journey.

Internet Fundamentals

In 1995, Bill Gates famously wrote a memo to Microsoft executives saying that the internet was now the "highest level of importance" for the company. He wasn't wrong, and today almost all software is, in some way or another, internet software. In order to understand how software works and how it breaks, you will need to know how the internet works.

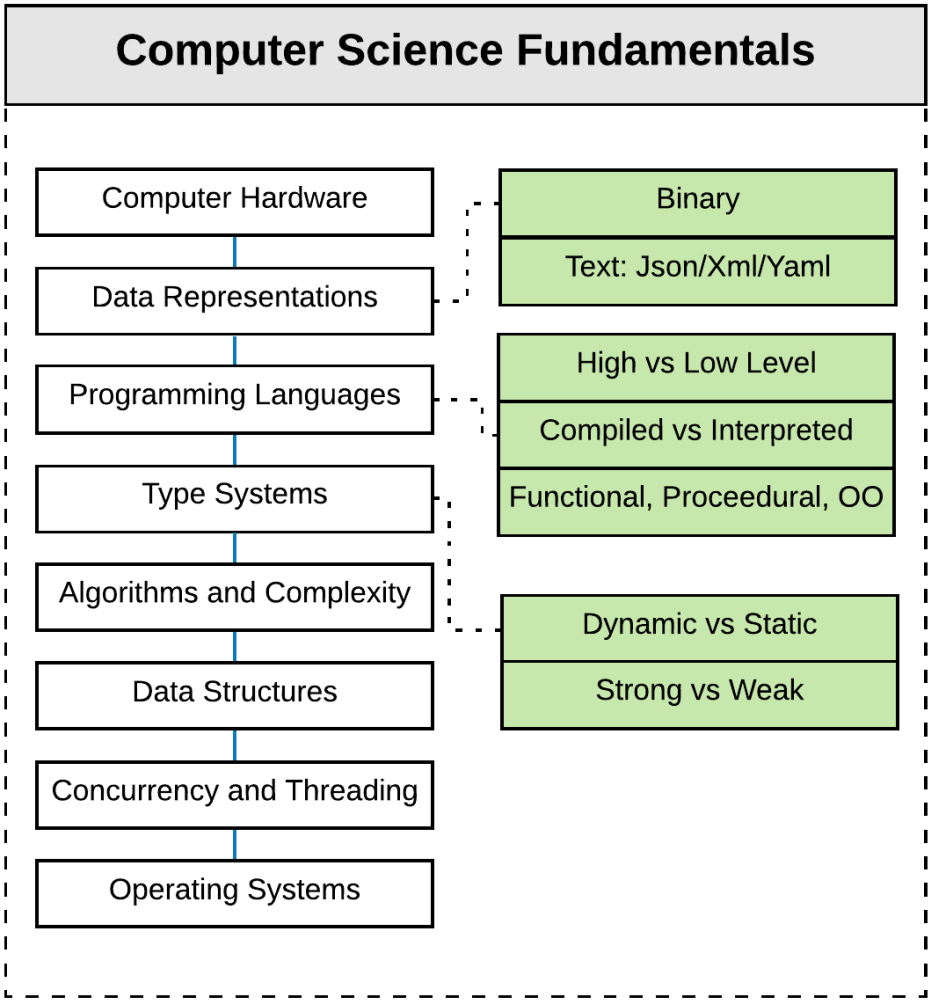


While this might seem daunting, don't let it dissuade you. You don't need a deep understanding of any one particular technology, but rather a high level understanding how the whole thing comes together. For example, you don't need to know that packet priority is set in the second 8-bit word of an IPv4 packet header, but you should know that IP ("Internet Protocol") relates to packets, and is what makes up the IP in TCP/IP.

Other important internet technologies you should be familiar with include things like DNS, HTTP, and the OSI model. Keep in mind the Web Application Fundamentals is a future category, so things like HTML/JS/CSS can all be put off till later.

While Internet Fundamentals will not be critical in your day-to-day quality engineering responsibilities, it does underlie many other important technologies, concepts, and tools. Learning how something like AWS Route 53 fits into a typical cloud application stack will be significantly easier if you have a foundation of internet fundamentals to build on.

Computer Science and Engineering Fundamentals



Ok, I can hear the shouts right now. “Computer Science! Computer Engineering! Those are four year university degrees, how can you expect me to learn all that on my own?” Don’t worry, there are several reasons why this doesn’t need to be scary.

1. There are many aspects of academic computer science and engineering that are just not relevant to quality engineering. For example, knowing what classes of computation problems can be categorized as NP, NP-complete, or NP-hard is an important topic in academic computer science, but won't help you much as a quality engineer.

2. While a formal education in computer science/engineering IS a good foundation for quality engineering, there is no magic guarded jealousy within the ivory towers of academia. Everything you can learn inside, you can learn outside, and almost all of it is available for free somewhere on the internet.

So what sort of computer science and engineering topics ARE relevant for the aspiring quality engineer? Subjects such as computer hardware and how it is controlled by operating systems and user programs. How these programs are written, and how different types of programming languages have evolved to do this. You should understand general characteristics of programming languages and how they can be categorized, such as high or low level, compiled vs. interpreted, functional vs. object oriented, and different approaches to type systems.

In addition to these foundational concepts, you should learn about algorithms and algorithm complexity and analysis, concurrency and threading, and other general concepts with programming. Keep in mind that programming itself (like, actually learning a language) is a later category.

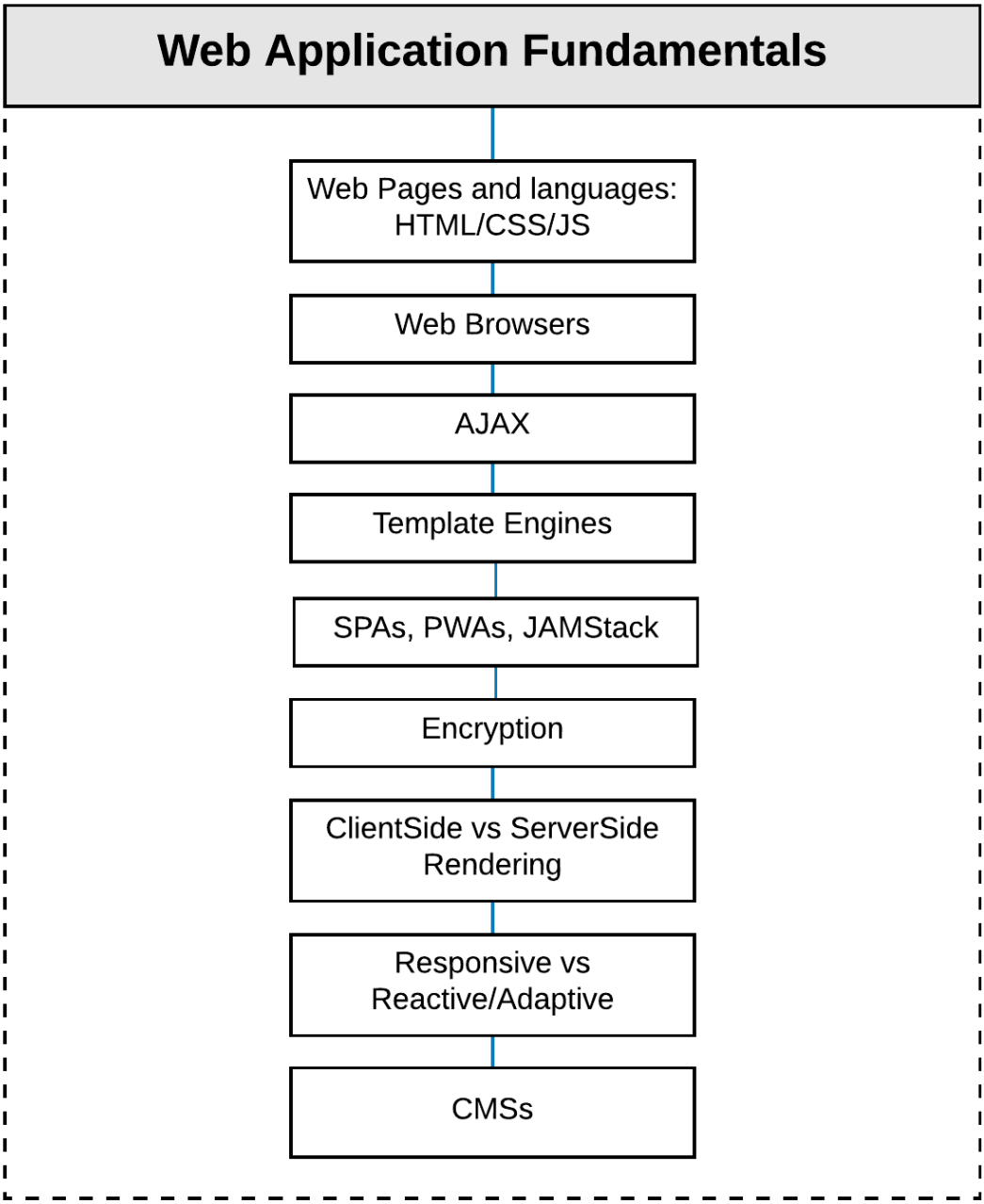
Some people might object to this overly theoretical start to computer science. Why is it necessary to learn type systems just to script automation?

First is the assumption that programming test automation is somehow less complex or requires less rigor than "real" programming. This is not the case, and we generally dislike calling the development of test automation "scripting" for this reason. Second is the assumption that you can learn a language by just skimming the top, rather than understanding the fundamental concepts on which all languages are built. In our experience, people with this deep understanding can pick up new languages more quickly than those that lack it. For example, I've known QAs who struggle with the concept of closures in JS despite using the language for years, whereas someone who understands the fundamentals of language theory would pick it up quickly and easily.

A quality engineer is just a specialized type of software engineer. A strong foundation in computer science and engineering will accelerate much of your subsequent learning; it will allow you to *understand* rather than just *know*. Don't underestimate the importance of this foundation simply because it doesn't seem immediately useful.

Web Application Fundamentals

Now that we have a foundation in computer science and internet fundamentals, we're ready to learn about web applications. While not all software you'll work on will necessarily have a web interface, web technology is so ubiquitous that special attention should be given to understanding how it works.

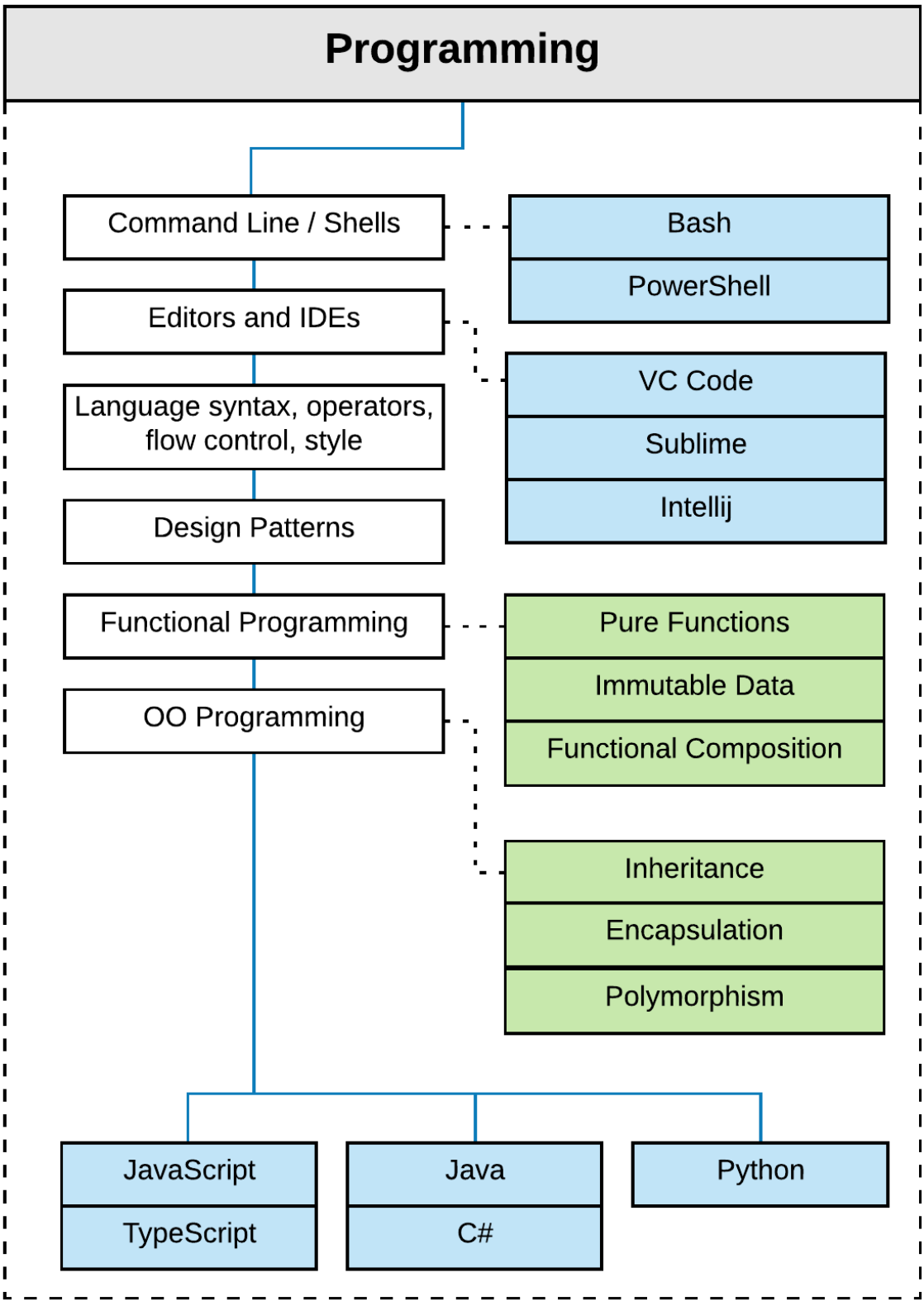


An acceptable level of understanding will include knowledge of how web sites are developed using technologies like HTML, CSS, and JavaScript. You'll need to understand how browsers work to take these languages and render them as an interface. Once you understand the basics of web applications, you can move on to do things like AJAX, single page applications (SPAs) and Progressive Web Applications (PWAs), followed by the likes of encryption, content management systems, and advanced topics like client vs. server side rendering and adaptive vs. reactive web applications.

Do you need to know how to build a web application from scratch? No, not really. But it wouldn't hurt. And having done it a few times in a few different frameworks (Angular, React, etc) will demystify the process and give much deeper insights into testing them. In addition, the trend in web technologies is for more and more of a web application to be testing "beneath the surface", for example, by mocking the data model and testing UI behavior as unit tests or with headless browsers. Building a few simple web applications will give you insight into how these different types of tests can be used and will inform overall test and automation strategy.

Programming

We previously talked about computer science, but didn't actually include programming. We know about type systems and memory and operations systems and such, but now we need to make that knowledge practical by learning to work in a programming language.



To be an efficient programmer, you should be comfortable with the environment you'll be developing in. This means understanding your integrated development environment (IDE) and

shell. When you program you should be thinking about the code, not how to find a particular button or setting.

In addition to the language, you should study the higher level constructs that organize code and inform good programming practices: design patterns, concepts like DRY and SOLID, etc. These concepts — and how they are realized in code — depend on the nature of the language you choose to learn, i.e. SOLID applies to object oriented language where as patterns like pure functions apply to functional languages.

The current programming languages we would focus on are JavaScript/TypeScript, Java or C#, and Python.

JavaScript is a great initial language as is it underpins all web development. In addition, it has been steadily encroaching on other areas of the enterprise through frameworks such as Node.js. Because of its use in web development, JavaScript is also heavily used in web automation through tools such as Protractor, WebdriverIO, and Cypress.io. JavaScript is continually ranked as one of the most popular programming languages.

While TypeScript is a fundamentally different language than JavaScript, we would lump the two together. Because TypeScript transpiles into JavaScript, much of the surrounding ecosystem is identical, and learning TypeScript in parallel with JavaScript will give you the benefit of having both a static-strong and dynamic-weak language in your arsenal.

After JavaScript/TypeScript, we would suggest either Java or C#. These languages are very similar and are both used extensively within enterprise application development. C# is slightly riskier, as companies that leverage C# tend to be "all in" on the .NET / Microsoft ecosystem, so you might not get exposure to much else. While Java and C# aren't new, sexy languages, they are the workhorses of the software industry.

Speaking of sexy (but not new!) languages, we have Python. Python has technically been around since the early 90s, but has seen a resurgence lately because of its use within data analytics applications, and specifically in AI/ML. Of course, it's also just a powerful and accessible language.

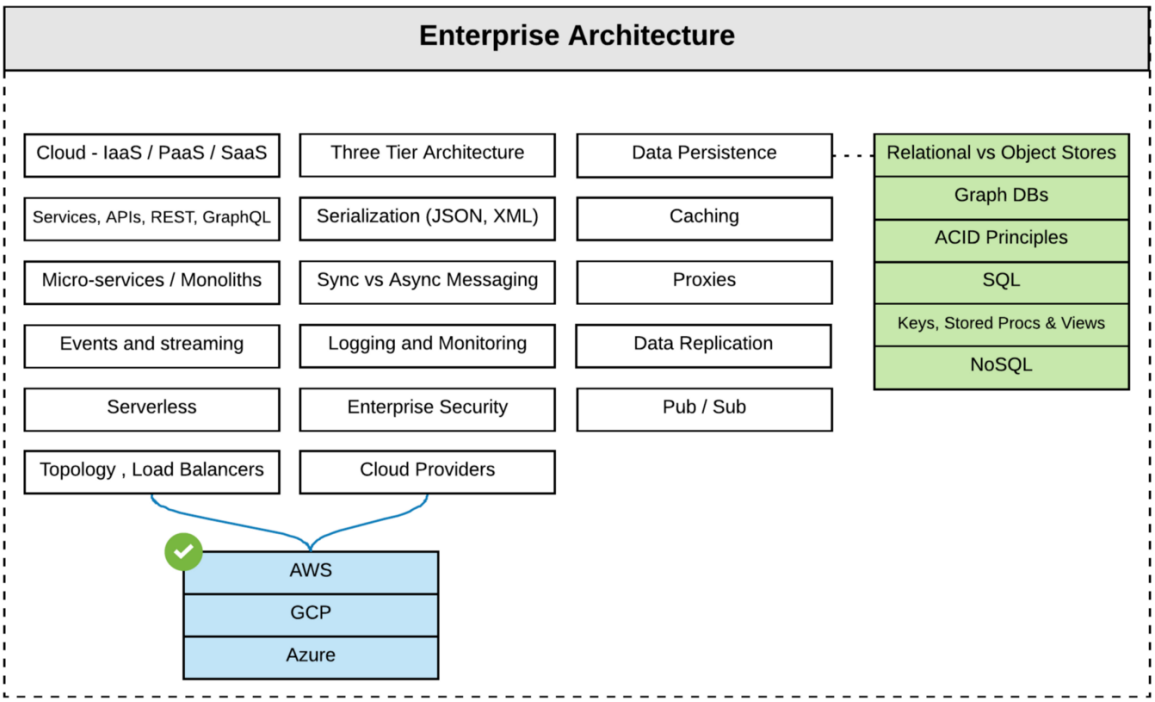
One word of caution as we talk about learning to program: programming is one of those things that is very easy to learn, but very hard to master. This often leads to confusion or conflict when people start their learning journey, or interact with people who are much further along. Sometimes you hear things like "I learned Python, but nobody will hire me. Not fair!".

Think of programming like learning to play chess: most of us could completely memorize 100% of the rules of the game in an afternoon. Does that necessarily make us grand masters? Hardly. In the same sense, you can learn the rules of a new programming language in a few weeks, but that doesn't necessarily make you qualified to call yourself a master. And it doesn't mean you are ready to step into critical application development roles. You'll learn all the rules in the first two weeks, but like chess, you'll spend the rest of your life trying to master them.

There are some in the quality assurance industry that say quality professional does not need programming skills. We disagree. We feel learning and continually improving in programming is a core competency of the quality engineer, and will be used extensively as you start writing complex test automation systems, as well when you collaborate with developers who are building the software you are testing.

Enterprise Architecture

Enterprise applications come in countless flavors, types, sizes, and configurations. In order to understand how these systems work — and how they break — you'll need to understand the basics of enterprise architecture.



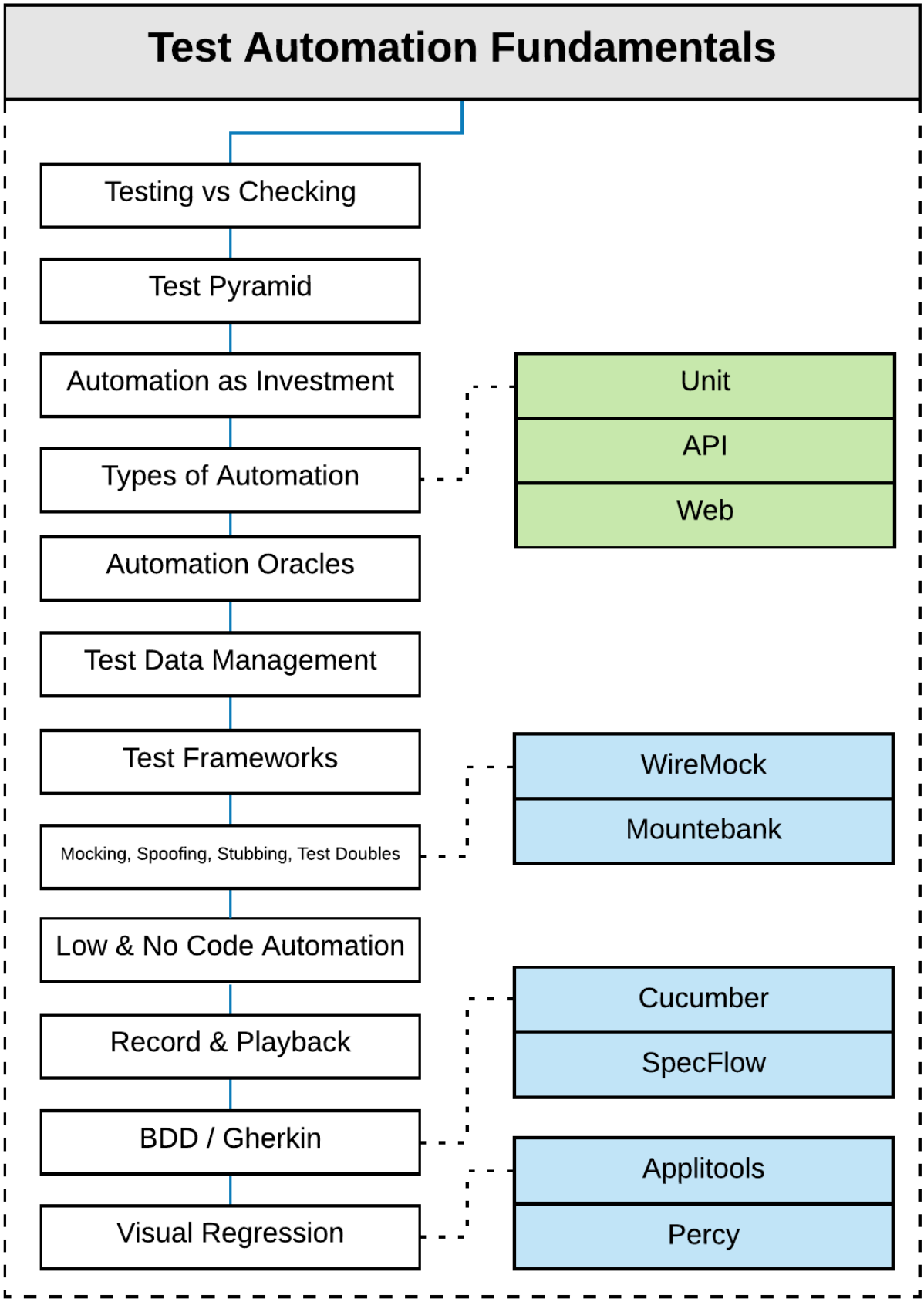
This is a broad category, but includes topics like three-tier applications, REST services, microservice architectures, streaming and event driven architectures, persistence strategies (relational vs. object stores, etc), replication, caching, proxies, etc. The topics in this section are not listed sequentially, as most are stand-alone and can be learned in any order.

In addition to understanding how enterprise systems are organized, you should also understand what they're built on. This means knowing IaaS, PaaS, and SaaS options, as well as a deep understanding of cloud offerings from the major providers like AWS, GCP, and Azure. Cloud technologies are becoming ubiquitous in modern architectures, so we cannot stress enough the importance of these topics. All major cloud providers provide extensive learning resources, and many third party learning resources (both free and paid) have grown to satisfy this demand.

Enterprise architecture is a vast field of knowledge, so focus on specific areas of study that are interesting or relevant to your goals. Some of these topics are more broadly applicable to many types of architecture (like persistence strategies), while others might only be applicable in some situations or in some roles (like event streaming). However, understanding the systems you are testing and how they are put together is important if you want to succeed as a quality engineer, as the expectations of this role go far beyond that of a tester, where black-box testing and a limited understanding of what goes on under the covers was fine.

Test Automation Fundamentals

Test Automation! Finally! There are many types of test automation — unit testing, API testing, etc. — and quality engineers need to have expertise in all of them. However, before we can go deep into any one type, we need to look at test automation as a theoretical concept.


Test Automation Fundamentals
Testing vs Checking
Test Pyramid
Automation as Investment
Types of Automation
Automation Oracles
Test Data Management
Test Frameworks
Mocking, Spoofing, Stubbing, Test Doubles
Low & No Code Automation
Record & Playback
BDD / Gherkin
Visual Regression
Unit
API
Web
WireMock
Mountebank
Cucumber
SpecFlow
Applitools
Percy

In order for test automation to be valuable, we need to understand why we write it. We need to look at test automation as an investment, how test automation can be described by concepts like the Test Pyramid, and how test data is reliant on things like test oracles, test surfaces, and test data.

We also need an understanding of how low or no-code automation fits into the picture, the benefits and drawback or record-and-playback tools, and how BDD languages like Gherkin are used.

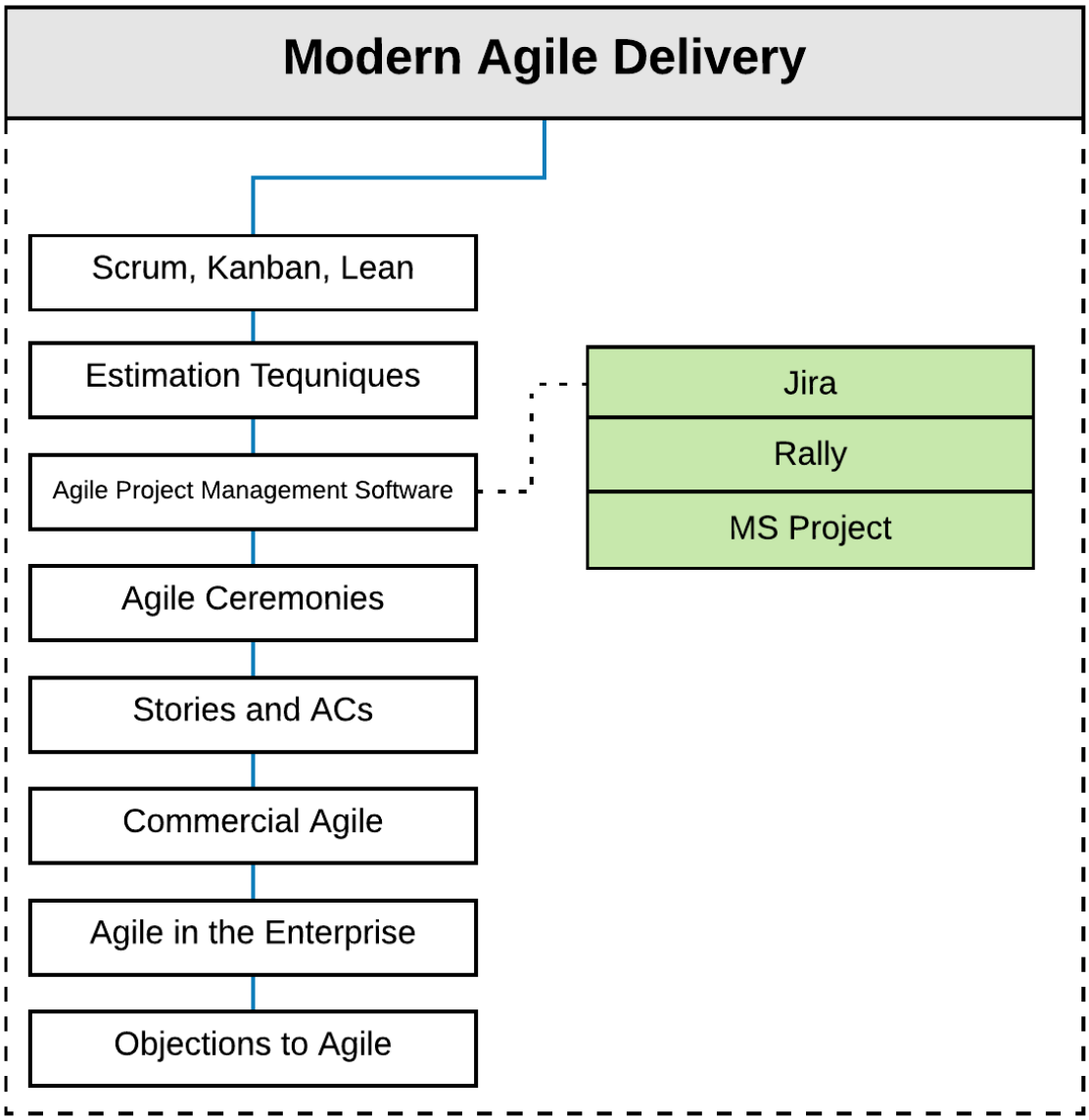
While we will go into detail later on unit, API, and UI automation, an understanding of what these types are and their respective benefits and drawbacks will be necessary in relation to things like the test pyramid and automation as an investment.

While we do not yet need to go deep into specific automation tools, knowing the categories of tools and some common examples in each category will be useful. For example, mocking or spoofing services to isolation the system under test is central challenge in most automation strategies, so knowing how things like WireMock or Montebank support this is helpful.

Test automation is a continually evolving subject, and there are some strong opinions and occasionally disagreement in this area. When this is the case, knowing both sides, rather than just what you are most familiar with, will make you stand out from other, less informed quality engineers.

Modern Agile Delivery

Before we dive deep into different types of test automation, we need to take a tangent into some less technical but still very important areas. While we previously looked at SDLC Fundamentals, we now need to specifically dive into agile delivery.



In order to operate effectively in an agile development team, you need to know more than just the theory of agile, you also need to be familiar with the tactical, practical aspects. And, while the best way to learn much of this is to experience it first-hand on an agile delivery team, it does not hurt to gain some understanding beforehand.

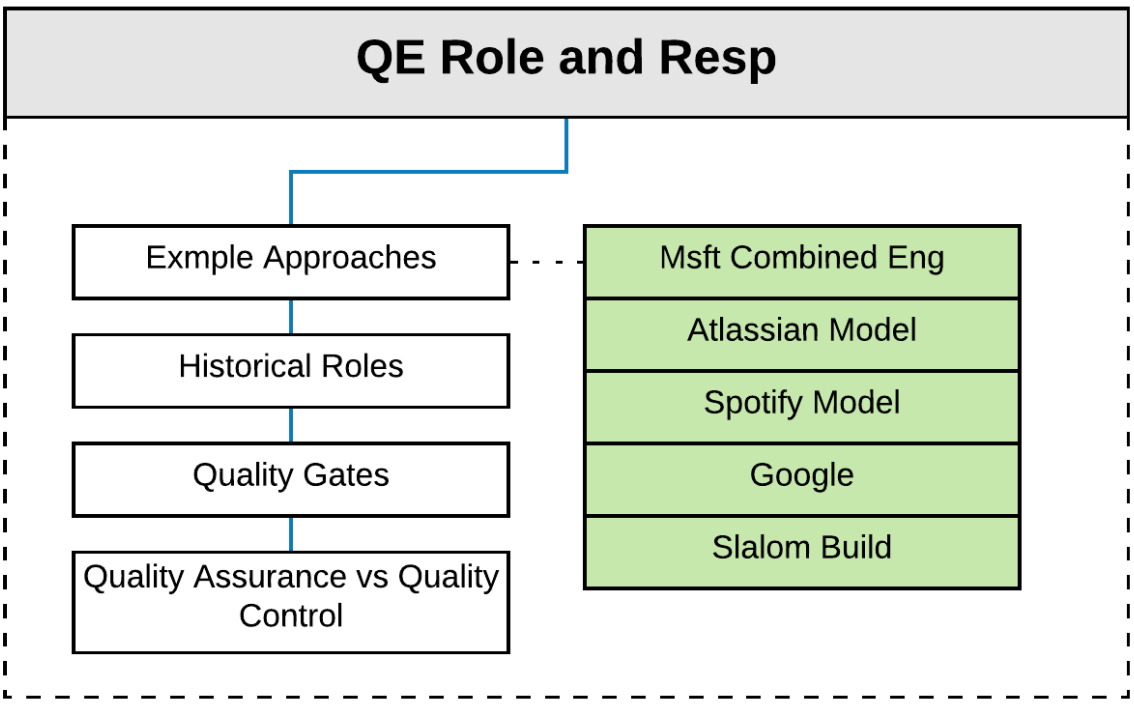
While every company does agile differently — the most commonly heard phrase about agile is an extremely exasperated "that's not real agile!"—there are many common approaches, ceremonies, and terms you should be familiar with as a quality engineer. They includes things like story definition, acceptance criteria definition, story estimation techniques, as well as the purpose of common ceremonies like stand-ups, retros, and showcases. In addition, it will be valuable to be familiar with typical agile project management tools. The most common by far

is Jira, but Rally, MS Project, and others also used. Plus, you'll get extra credit for familiarizing yourself with applications of agile in the enterprise like Scaled Agile (SAFe), LeSS, or Nexus.

Knowing how your agile team operates and being able to identify when it's working well, and when it isn't, is critical to the role of quality engineer — often it is these challenges that are the root cause of software quality issues down the line.

The Quality Engineering Role

What exactly does a quality engineer do? Where does this role end, and that of a software engineer begin? How and when does the quality engineer collaborate with all the other roles of an agile development team? It's important to have clear and explicit answers to these questions before beginning.



Unfortunately, the answers to these questions not only differ company-to-company, but also team-to-team. The skill-sets, domain, expectations, timeline, and culture of every team and company will influence what a quality engineer does, so this will have to be a higher-level, theoretical treatment. Regardless, knowing your role will be extremely valuable in being successful.

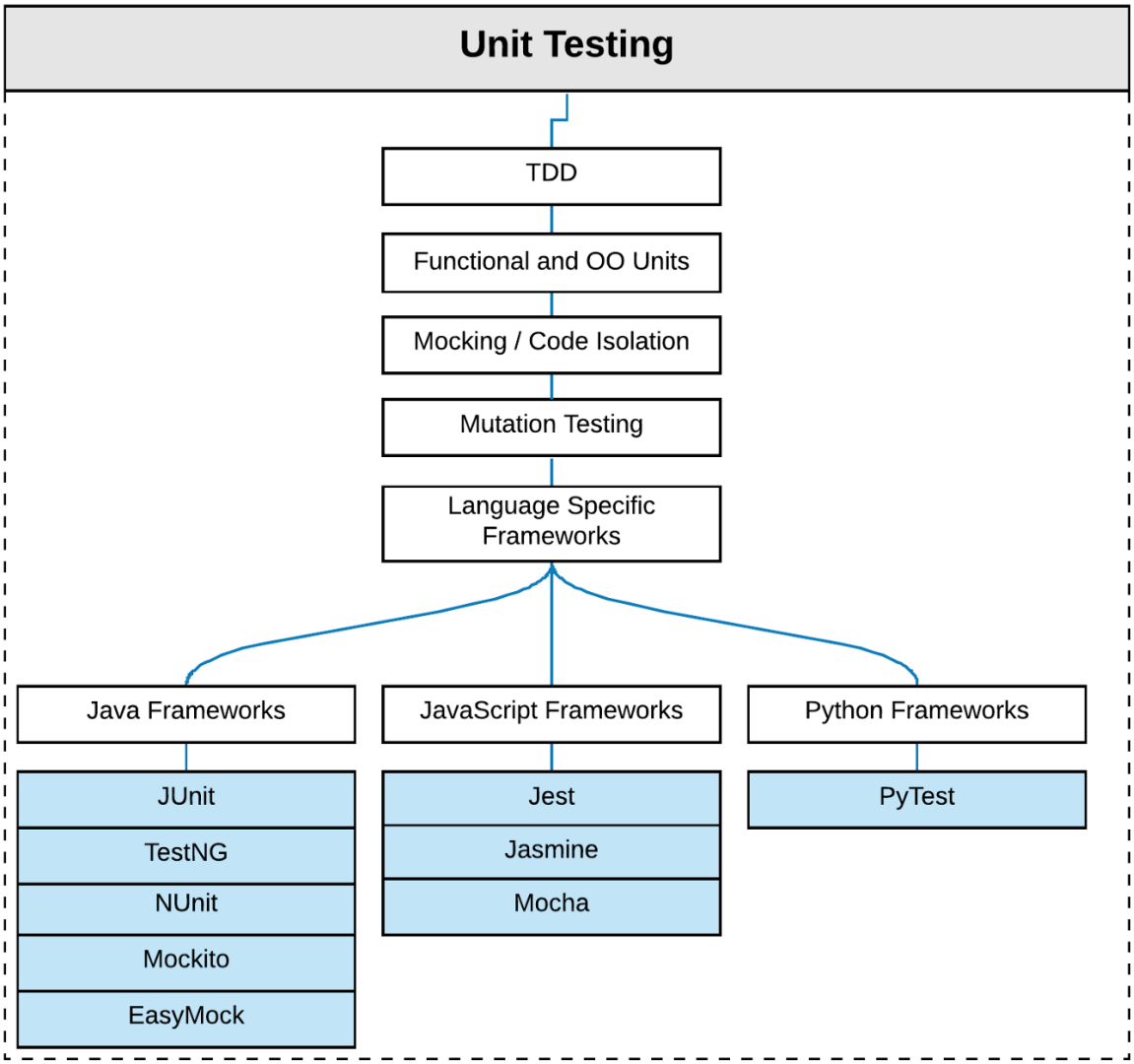
Regardless of how your specific company operates, it will be valuable to have a broad understanding on how different model technology companies approach a problem. For example,

check out How Google Tests Software, read about Microsoft's Combined Engineering Approach, Atlassian's Quality Assistance, the Spotify Model, and especially (self plug!) Slalom Build's Quality Engineering Core Principles.

Unit Testing

While unit testing is usually performed by software engineers, quality engineers should have deep expertise as well. Quality engineers should know the tools and frameworks, know the drawbacks and pitfalls within their particular language and application, and know exactly how unit test coverage augments and supports other higher-level layers of automation. Quality engineers should have no problem code reviewing unit tests, or even implementing the tests themselves.

The level of expertise we require in quality engineers for a type of testing mostly implemented by software engineers might surprise you. However, this understanding is critical given the role and expectations outlined in previous sections. Quality Engineers are not just testers or automators of tests, they must think holistically about application quality and understand everything that might impact that quality. Unit testing makes up a large and important feedback loop for development teams, and quality engineers need to be able to collaborate as peers with software and devops engineers on implementing and maintaining the health of these tests.

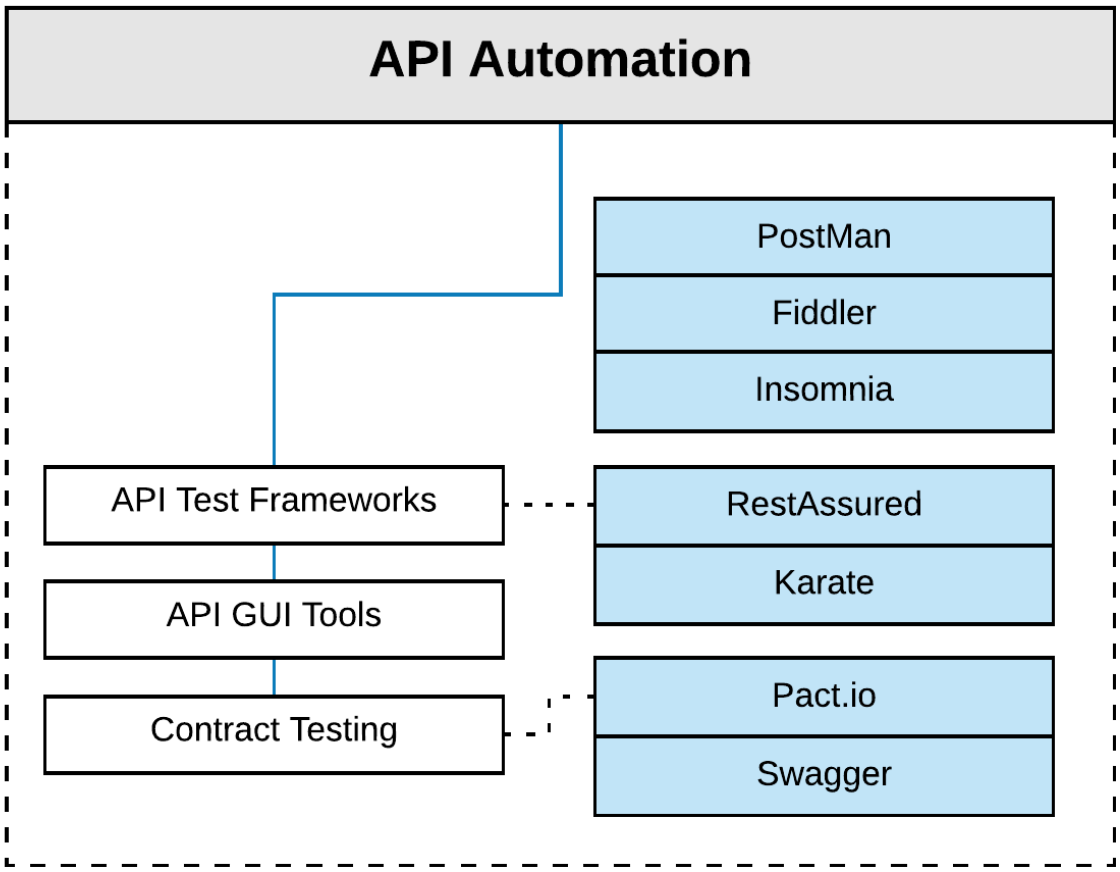


Within unit testing, you should understand and be comfortable leveraging TDD to develop unit tests. You should understand the differences between unit testing a function vs a object oriented language, and the patterns that make unit testing easy. While most unit testing frameworks support very similar functionality, there is enough difference to make learning multiple within the language you are focusing on valuable.

While this area is focused on unit testing, automation professionals will know that test automation doesn't necessarily fit into clean layers, and should be thought of as a continuum from discrete and small tests to large and expansive tests. Knowing how all of these types of tests come together is critical for developing a cohesive automation strategy, and is another reason why quality engineers must have deep knowledge and understanding of all types of tests, even if they aren't necessarily the ones authoring them.

API Automation

API automation is another type of feedback loop that is critical for any non-trivial software system, and one that all quality engineers should have a deep understanding in. While what constitutes an API is a bit flexible, we are usually talking about REST services, web services, SOAP services, topics and queues within streaming architectures, file interfaces, and maybe even lower-level binary protocols.



APIs are built to be consumed by computers (it's in the name!), so it's no surprise that they make good test surfaces for test automation. Because of the limitations of both unit and E2E / UI tests, quality engineers need the ability to build, execute, debug, and otherwise maintain these types of tests.

In addition to API *automation,* tools like PostMan that support ad-hoc API testing are also valuable. Most common API automation development workflows will contain a significant amount of API interrogation with tools like Postman while building actual tests in a programming language.

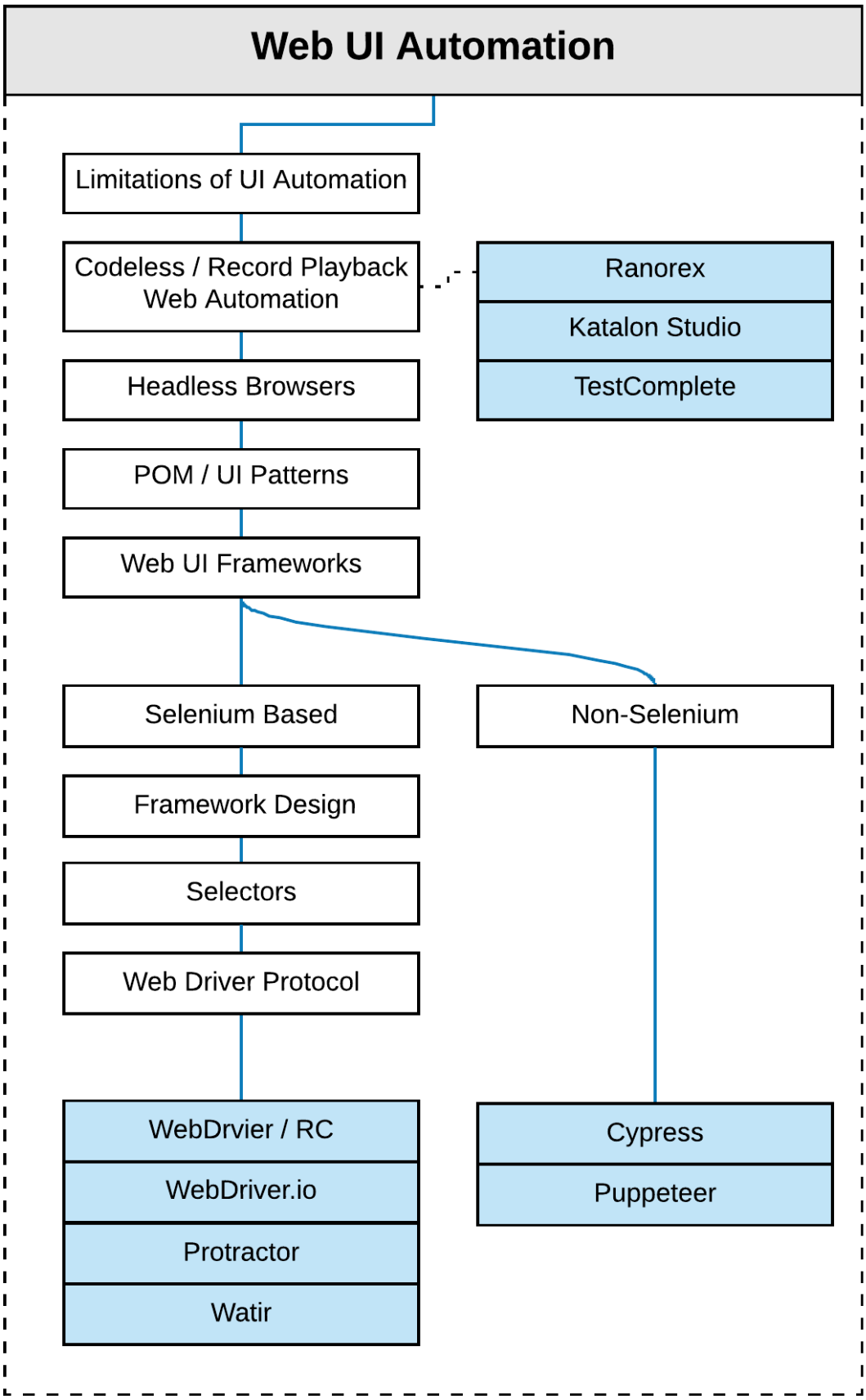
While we have called out two common API frameworks: RestAssured and Karate, API testing can be achieved with just a test runner (like JUnit) a library to interface with the service

(like HttpClient) and an assertion library (like Hamcrest). Every programming language has these.

Depending on your team the responsibility for API tests, like unit tests, might be a developer responsibility. Regardless, you will need to be very familiar and highly comfortable building, extending, debugging, and executing API automation.

Web UI Automation

UI automation is, unfortunately, what most people think about what they think of automation. While we know that it only makes up one layer of our overall automation pyramid (and the smallest at that) it still plays a significant part in the larger automation strategy. If your application has a UI (and most do), automating the UI is the only way to create a true end-to-end test, and it is the end-to-end test that reflects the closest approximation of end user experience.


Web UI Automation
Limitations of UI Automation
Codeless / Record Playback Web Automation
Ranorex
Katalon Studio
TestComplete
Headless Browsers
POM / UI Patterns
Web UI Frameworks
Selenium Based
Non-Selenium
Framework Design
Selectors
Web Driver Protocol
WebDrvier / RC
WebDriver.io
Protractor
Watir
Cypress
Puppeteer

Unfortunately, it is also one of the most challenging types of automation. While APIs are created to be consumed by applications, user interfaces are created to be consumed by… well… users. Asking an application (like test automation) to drive something built for users is often like driving a round peg into a square hole. To deal with this, a huge number of tools, frameworks, and automation applications have been created to help out.

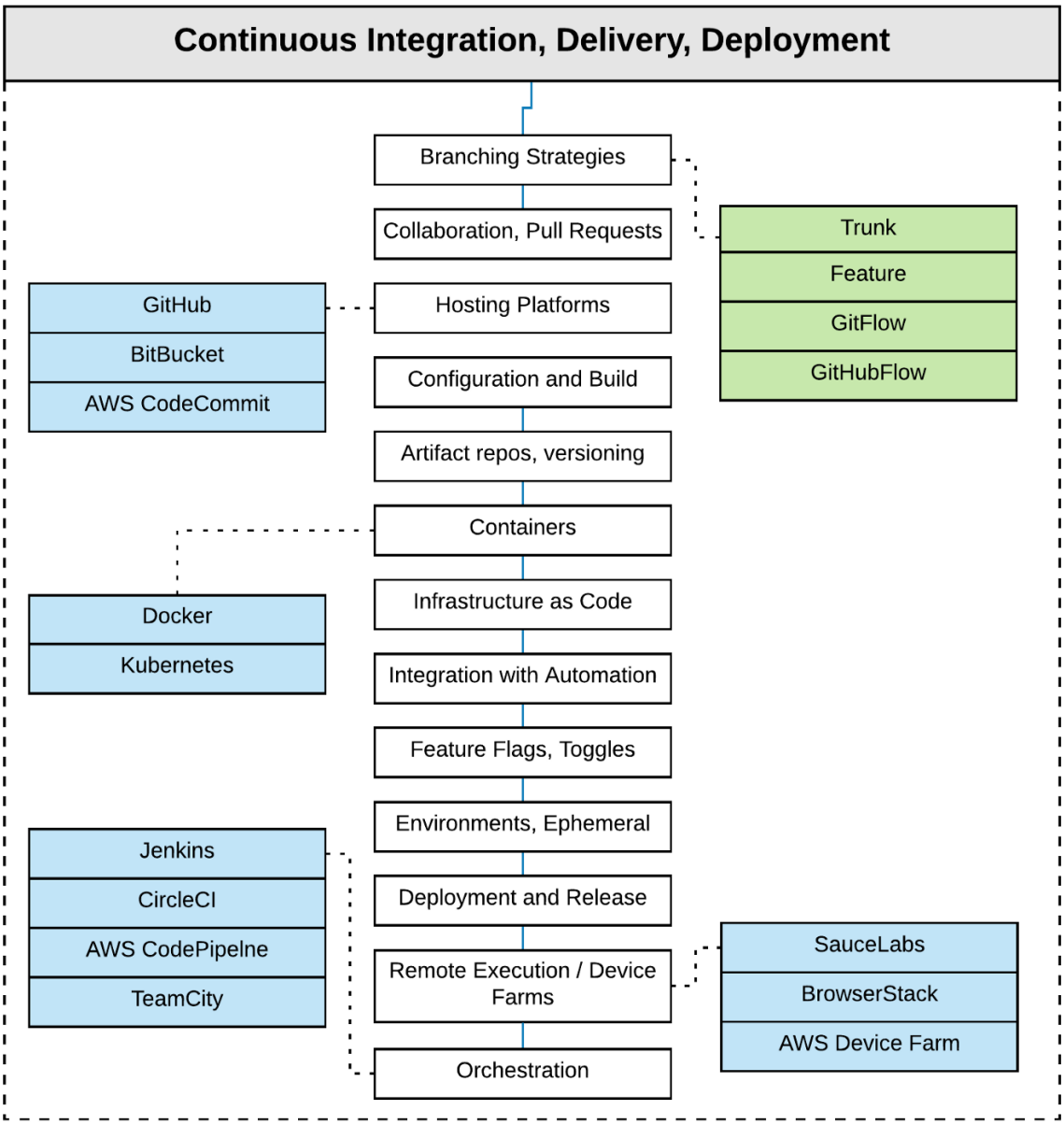
The most common of these, by far, is Selenium and its wrappers and derivatives (WebDriver.io, Protractor, Appium, etc.) There are too many to learn all of them, but instead, strive to understand what Selenium is doing and how it deals with the challenge of driving web browsers. Once you understand this and have a command of the programming languages you will be using, picking up the next trendy UI automation framework based on Selenium won't be hard at all.

In addition to Selenium, there are browser automation frameworks that attempt to drive web automation without using Selenium's underlying WebDriver protocol. These tools usually interact with browser-specific APIs like chrome's DevTools, and can make UI automation for these browsers significantly easier and more powerful, albeit with some drawbacks. Tools like Puppeteer and Playwright fall into this category. Cypress.io is another new and promising tool in the non-selenium category that is worthy of learning. Again, the tools themselves are less important than knowing how web automation works, combined with foundations in web architecture and programming will allow you to pick up any new tool quickly.

In addition to selenium and "not selenium" UI automation, there are many commercial, proprietary automation tools. They're usually marketed to the less technically savvy, and can be a valuable for doing simple things. Knowing when they are appropriate and when they are not, and being able to see through a lot of the marketing material (We use AI to automate everything at no cost!) is important for understanding their role relative to more code-centric approaches.

Continuous Integration, Delivery, and Deployment

Test automation is most valuable (some would say it is ONLY valuable) when it automatically provides direct and immediate feedback to all system changes, which means integrating test automation with continuous integration and continuous delivery/deployment pipelines. Tests that sit in a repo and are only run when someone clicks them quickly rot and are inevitably thrown into the dust bin. As a quality engineer, you will need to understand the concepts of CI/CD, be comfortable with the tools involved, and be able to collaborate with devops engineers and software engineers to implement an overall test automation approach that is valuable to everyone.

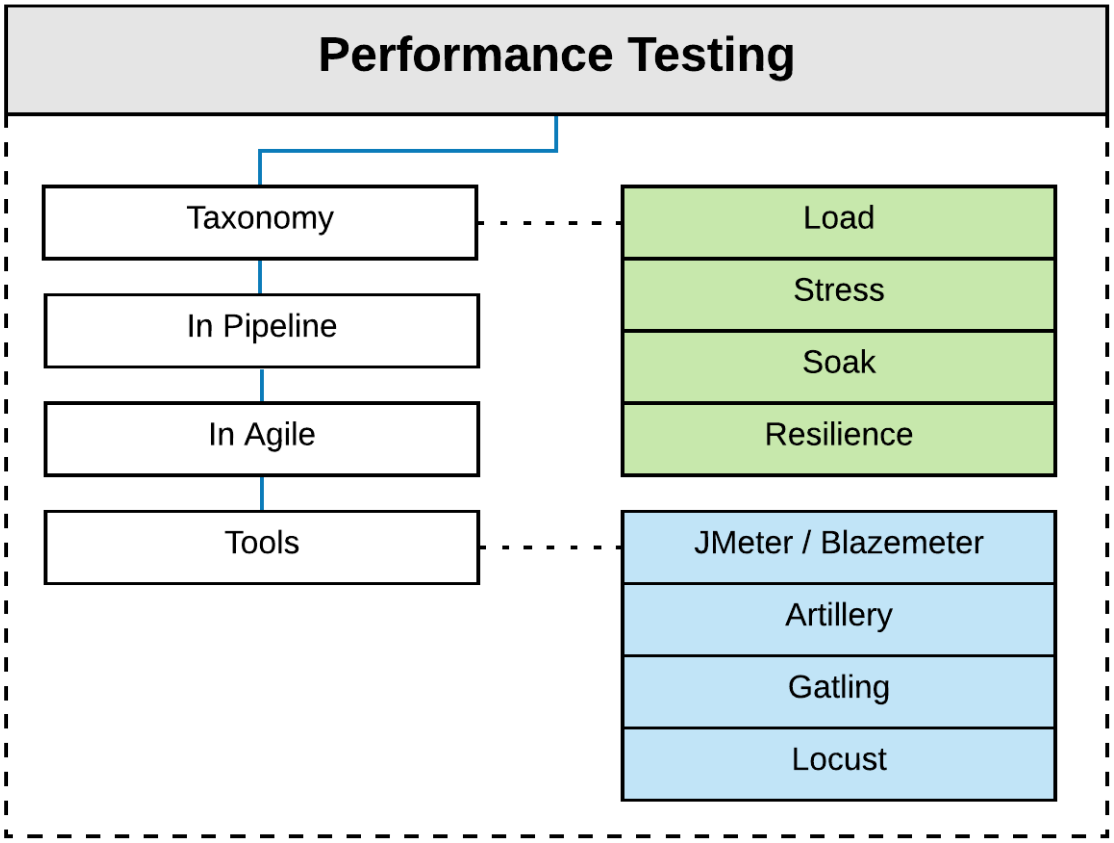


How far this understanding goes into implementation will be different on every team, but from our experience, it is not uncommon for quality engineers to be directly involved with implementing pipeline infrastructure. This might mean writing Jenkins pipelines, or working with Cloud Formation scripts, or some other technology leveraged by your team. Regardless of what is used, quality engineers should not be intimidated and should be ready to assist or support all CI/CD activities.

Your CI/CD strategy will necessarily overlap with your test environment strategy, and this is another area that quality engineers should be familiar with. What are gating checks, when should shared environments be used, what is the dependency between test environments and test data management? Quality engineers should be ready to answer all of these and more.

Performance Testing

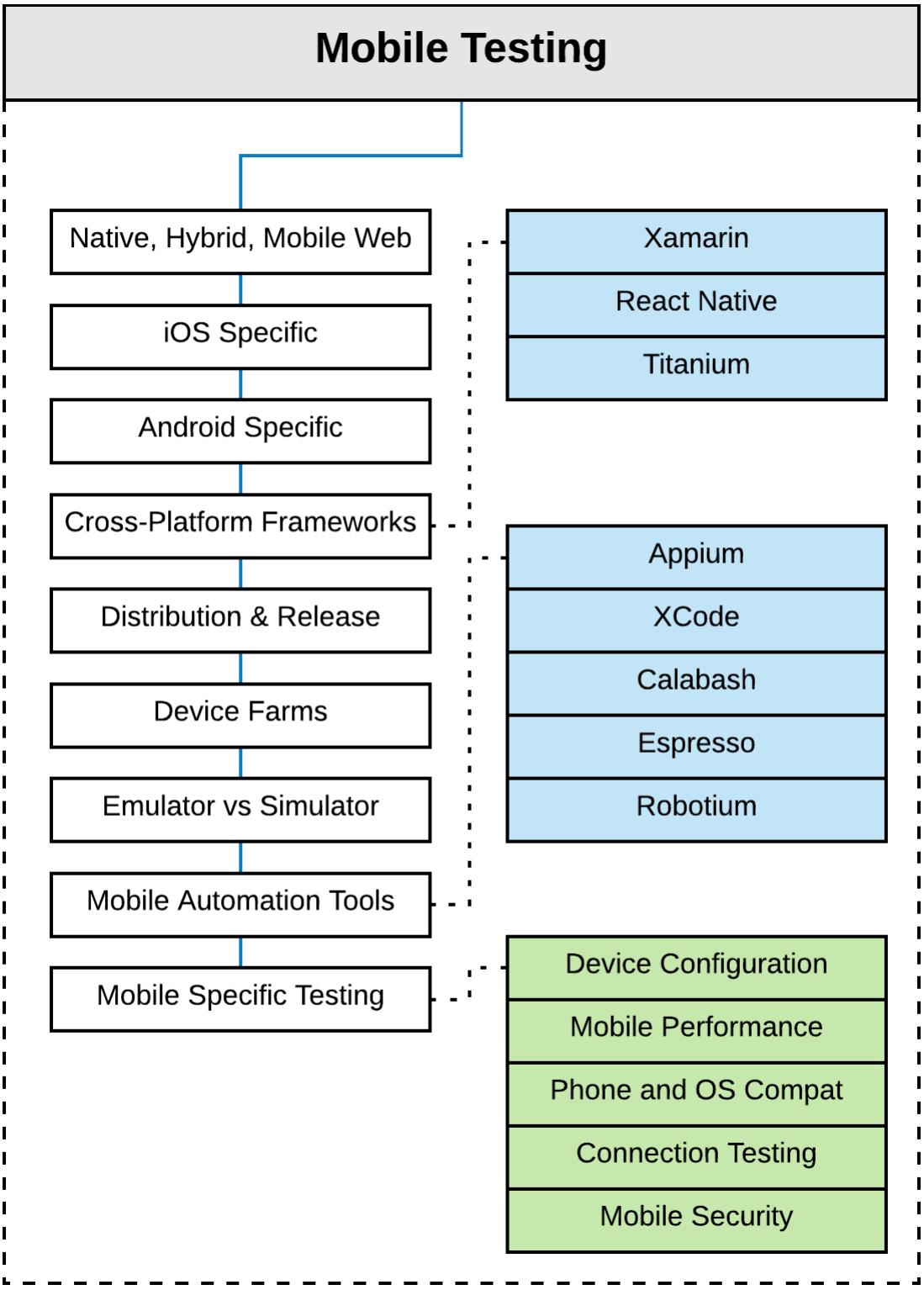
It doesn't matter if applications work if they are too slow to be used or suffer timeouts under load. Performance testing ensures software meets performance expectations, and is a critical area of expertise for quality engineers.



While performance testing is a deep and broad topic in itself, and there are test professionals who specialize in nothing BUT performance testing, you should understand the types and nomenclature of these tests, the tools that support them, and how these tests can be integrated into CI/CD pipelines and agile processes.

The reigning champion of stand-alone performance testing tools is Apache JMeter, but wide variety of both code-first and GUI based tools exist. Like other areas of test automaton, we recommend learning tools aligned to the tech stack you are most interested.

Mobile Testing



Mobile internet usages has surpassed desktop usage, so testing mobile interfaces is no less important than testing web interfaces. Within the "mobile" category, you should understand the nuances of testing native apps, hybrid apps, and mobile web interfaces. Within the native app

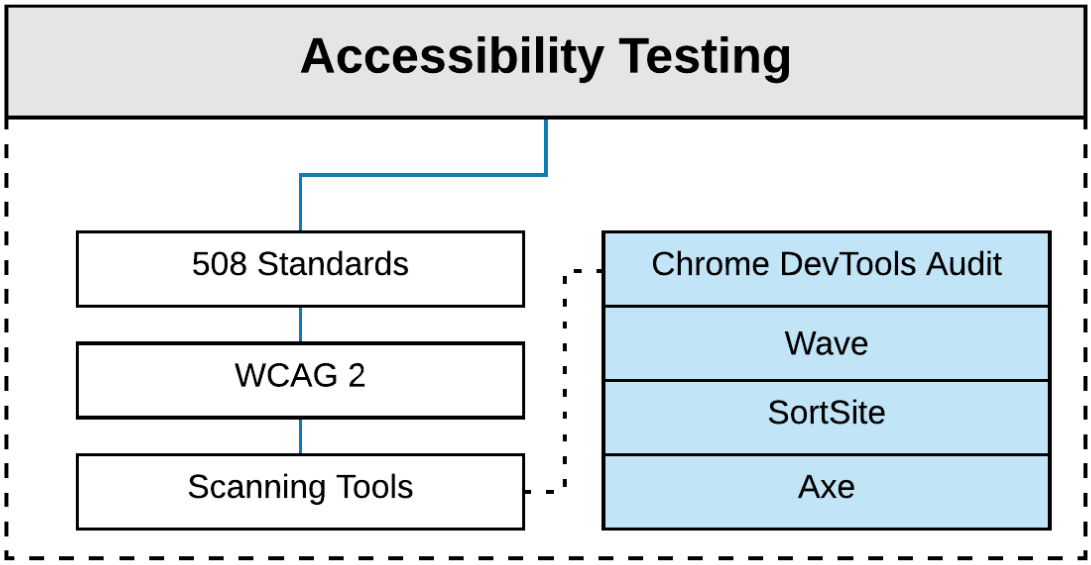
category, you will need to know how to test both Android and iOS platforms — because of course there are differences.

Automating mobile applications presents it's own set of challenges, and an plethora of tools and frameworks used to overcome them. There are platform specific tools like Espresso for Android and XCUITest for iOS, but also cross platform tools like Appium. Just like web automation, we recommend learning tools that leverage a code-first philosophy, rather than GUI-centric tools.

In addition to mobile automation, you should understand how device farms (both on-premise and cloud) can be used to expedite testing across the huge number of devices types and OS versions. We should understand how emulators and simulators can be used to get test feedback without physical devices, how mobile application distribution and release differs from other types of software, and other test areas specific to mobile applications that would not be found testing normal web based software.

Accessibility Testing

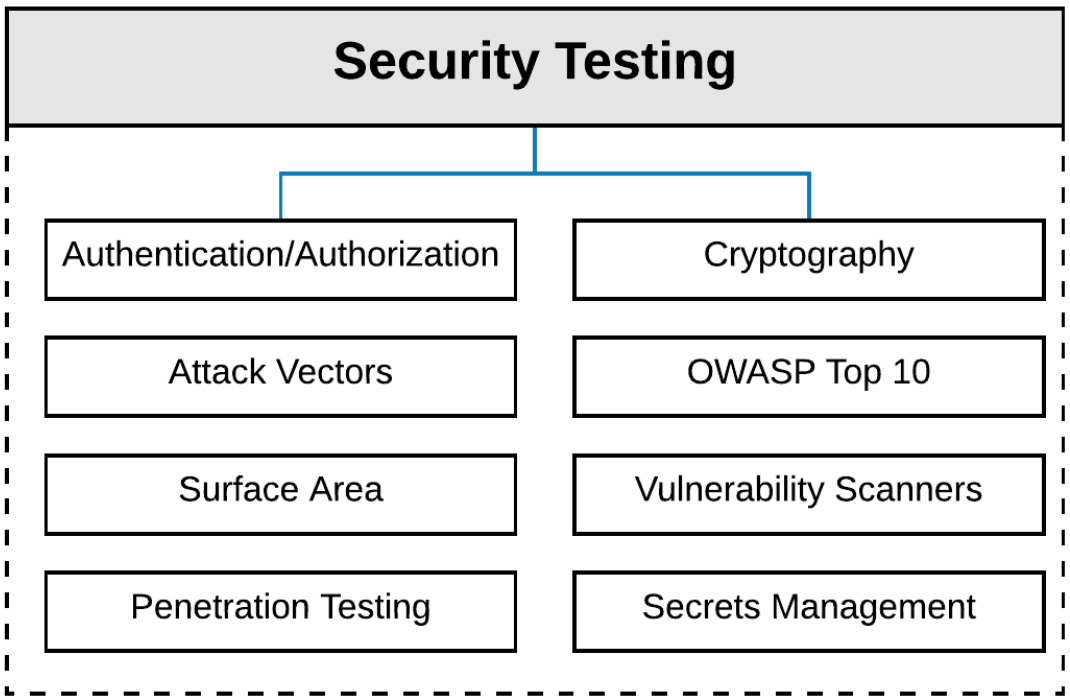
Just like all commercial services, software must be accessible to those visual, hearing, or other impairments. As quality engineers, we should understand these requirements and the methods and tools used to validation them.



In the United States, accessibility is defined by the government's 508 Accessibility Standards and the Web Content Accessibility Guidelines (WCAG) defined by the W3 consortium. If you are going to be working on web or mobile interfaces, understanding these standards and the plethora of scanning tools used to evaluate compliance is important.

Security Testing

Software security is incredibly important to the overall quality of software systems, so why does this topic appear relatively late in our learning roadmap? The importance of security, and the highly technical nature of most security testing has pushed the industry adopt specialized security roles; for example: network security, protocol security, enterprise security, etc. As a new quality engineer, you probably won't have involvement in any of these. However, you should still know the basics of security, including how to secure the process of building software itself.



This includes concepts such as how authentication/authorization is used to limit access, the Principle of Least Privilege, how cryptography works (eg, public key encryption), and an understanding of OWASP Top 10. You should understand security terminology like attack vectors and surfaces, how some vulnerabilities can be identified using scanners, and the basics of penetration testing. Unless you decide to pursue a career specifically in software security, these topics should be enough to get you started as a quality engineer.

Extra Credit: AI/ML Testing and Data Engineering

AI/ML and Data Engineering are both hot topics in software development, but we decided to omit them from the core roadmap because 1) this is already rather long, and 2) these are specializations that might not be applicable depending on the role and company you end up at. However, if you'd like to go down this path, the foundation you developed in previous sections

like enterprise architecture and automation fundamentals will serve you well, and it will be no problem to add AI/ML or data engineering testing to your testing toolbox.

Summary

This size and scope of this learning roadmap can be intimidating-144 topics across 18 sections, some of them are quite deep! Take the breadth of the roadmap as an indication that a career in quality engineering can be rewarding if you love learning, enjoy technology, and like solving complex problems. With command of these topics, the value you can bring to an organization is enormous.

If you are already in the quality field and can think of something we missed, feel free to say so in the comments below. While we spent a lot of time developing this, I guarantee things were overlooked. If you are just starting out in this career: Welcome! And good luck on your learning journey.

Special thanks to Kelsey Davis, Snehal Lohar, Lindsey Driscoll, Jason Hill, Jason Varland, and Nader Hatami for their feedback