**13. Enterprise Cloud Native Software Engineering**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Aff2)

(1)

Bangalore, India

As a software engineer, you get to see your work being used in every aspect of life, and you are responsible for how it is developed and deployed in an environment. According to recent global surveys, a large percentage of software deployed around the world is poorly designed and executed and the people using it are unaware of the socially engineered risks of software exploitation. Software is not just about an executable deployed on a server; it is about adaptability, agility, accessibility, security, and intelligence.

A methodology is a set of guidelines and principles that can be tailored to a specific situation. It can also be a specific approach, with templates, forms, and even checklists used throughout the project or product lifecycle.

**Note**

Consulting organizations use the term *project* for a work product delivered to enterprises.

Methodologies enable you to implement shared experiences and a ubiquitous language across diverse teams located geographically.

Agile methodologies introduced principles that put software development, quality, and collaboration above contracts and plans. This improved software quality and development approaches because now customers are involved in the requirements phase and development phase with a clear definition of what the business and customer wants.

In previous chapters, you learned how to create a cloud native architecture and design cloud native systems. In this chapter, I will explain the methodologies you can use to develop cloud native systems. There are tons of books and articles on software engineering, which I am not going to rewrite or explain here. In this chapter, I will cover the details of how the software engineering methodologies relate to cloud native systems. In addition, I will explain how you can enable a culture of agility by using these development methodologies.

In this chapter, I will cover the following:

* Distributed agile methodology
* Feature-driven development
* Hypothesis-driven development
* Test-driven development
* Behavioral-driven development

**Cloud Native and Traditional Application Engineering**

The nature of the application development process has changed in recent years, especially with the evolution of cloud native services. This affects agility and automation of software engineering and, cloud native has pushed enterprises to adopt modern day technologies and process. This adoption of modern technologies help enterprises to meet the customer expectations and to develop faster to market.

This transformation in development methodology helps to move from the traditional to cloud native application development mindset and process.

The difference between traditional and cloud-enabled development methodologies is vast, and the application development process in a cloud environment provides larger operational and economic benefits by using platform as a service (PaaS).

Whether you are modernizing a legacy system or building a new one, you are likely using a PaaS environment. Enterprises are rapidly finding that cloud native application development offers numerous benefits over the traditional approach.

Cloud native is about how applications are developed and deployed, not where. Innovative software engineering is required to leverage all the benefits of cloud computing and mitigate its challenges strategically to push forward its advances.

As I mentioned in earlier chapters, cloud native systems are built upon using the cloud native elements with DevSecOps and agility, collaboration across global teams, and stakeholders.

The priorities of cloud native application development over traditional development are as follows:

* Speed to market, with an emphasis on quick turnaround of new services or changes to existing services. This is compared to slow traditional development methods, long-term development plans that assume underlying binaries, and business value that remains stable for a longer duration of time.
* Short, continuous development cycles, rather than long timeframes.
* Key pod culture with Conway’s law compared to traditional resource mapping.
* Built on containers and services infrastructure utilized both in the cloud and on-premises rather than server-centric based on VMs and bare-metal servers.
* Feature-driven, hypothesis-driven, model-driven, test-driven, behavioral-driven compared to traditional requirements-based modeling.

**Intelligent Software Engineering**

As you know, 20th-century engineering is a thing of the past, and 21st-century engineering is evolving rapidly in all sectors of industry. For example, civil engineers have moved from building over nature to designing within nature, and industrial engineers have to consider the end-to-end lifecycle of their products from concept and design through manufacturing and service support. Today’s engineering processes must be connected to the pulse of the customer’s every move and understand them in real time. Design your application for outcomes and focus on driving velocity to value.

Engineering capabilities must help your clients with the following in mind:

* Creating new markets, not increasing market share
* Small, autonomous, self-organizing “2-PIZZA” teams with a culture of innovation, culture of governance, and culture discipline
* Creating automation to bring features to market faster
* Using cloud platforms
* Focusing on customer success and tying this to rewards
* Implementing a culture of go, not consensus: test, learn, repeat
* Delegated decision-making to the level of individual contributors
* Distributed code ownership to avoid perverse incentives

Table [13-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Tab1) illustrates the differences between traditional and cloud native. In the cloud native world, you need to use intelligent engineering for your customer, and you need to have the right mix of methodologies to follow to develop a cloud native application.

***Table 13-1***

Traditional vs. Cloud Native Engineering

| Traditional Software Engineering | Cloud Native Intelligent Engineering |
| --- | --- |
| Product centric | Customer centric |
| Focus on predictability and efficiency | Focus on speed to value and innovation (fail fast) |
| Co-located teams | Distributed teams across regions and geographies |
| Design stand-alone products | Connect to an ecosystem of products |
| Structured, linear process | Faster, customer-centric, modeled agile process |
| Large batch deployment | Lean product management, push-based deployment, and single-click deployment (MVP) |
| Highly skilled, manual coding, testing and deployment | Automated coding, testing, deployment |
| On-premises development | Cloud-enabled development |

The following are a few methodologies we’ll cover in this chapter:

* *Hypothesis-driven development*: In the modern-day user experience, you need to create a prototype, test, and rebuild services and the user experience until it’s acceptable by the user. Hypothesis-driven development helps you to develop an application by involving end customers.
* *Behavior-driven development*: In present-day architecture, you are building for business, and you cannot complete your system architecture and development without the active participation of the business. The behavior-driven development helps you to develop an application by collaborating with stakeholders.
* *Feature-driven development*: For modern-day applications, you need to develop an application that is customer-centric, iterative, and incremental, with the goal of tangible efficient software results. Feature-driven development helps you track progress and results.
* *Test-driven development*: Writing test cases first before developing the code helps you to improve the quality of software. Test-driven development converts requirements to functional test cases. This helps provide more clarity for engineers to craft their code.

**From Project to Product**

The objective of every project is to deliver customer value, but how you deliver the customer value is most important. In projects, customer value has start and stop dates, which means the team stops supporting the project once it closes. Indirectly that is the end of the customer value, but it doesn’t have to be.

The product mindset is to have continuous delivery of customer value, where there are no end dates and the team continues to support the customer value. In today’s world, there is a dramatic shift from project to product.

A project is temporary in that it has a defined beginning and end, a defined scope of resources, and a defined set of requirements, and it doesn’t address larger market share and new markets. On the other hand, a product is a good service, platform, or application that is created generally for sale to meet customer and business needs.

In a present-day cloud native architecture, major mindset shifts are required to move from executing projects to maintaining products, because of end-user expectations. Over time, your client might enter into a new market with a different culture and a different set of users and need to change how they engage a new set of users with business needs that change in response. If you have a project mindset, you might not have a historical record because resources are moved to other projects that need to address new user expectations.

**Organization Transformation**

You may have experienced many projects that have delays, low quality, no collaboration, or other issues. These kinds of projects have common problems such as the following:

* *Code changes*: This is the fear of making changes to the existing code, as the non-cloud native solution is hard to maintain or evolve or so fragile that the slightest changes can have a significant effect on the whole system.
* *No appropriate automation in the software delivery cycle*: This leads to a lot of manual effort that is needed to get new use cases into a working application deployed into an environment.
* *Nonproduction environments are limited and require more time to provide them*: This causes development efforts to be inefficient or partially wasted.

A few matured organizations are experimenting with new modern-day software engineering practices successfully and with great results. Companies like the Guardian, Netflix, Google, and Amazon can put new software into production in near real time by making small independent changes and a fully automated process of the development lifecycles. These changes impact only a small part of an application at a time. With all these changes, the companies save millions of dollars a month. For example, Netflix uses chaos engineering. This allows Netflix to proactively identify and resolve platform issues in production before they impact the customer. In addition, Sportify can create a whole agile pod team organization/culture that is inspiring others. These success stories help us define modern engineering.

The following are the benefits of embracing modern engineering:

* Faster to market; better user experience with hypothesis-driven development and the ability to transform new ideas into business value
* On-demand provision of environments; eases the nonfunctional testing
* Single-click deployment with continuous integration and delivery and infrastructure as a code
* Use of available PaaS services across cloud providers by using model-driven development
* Decentralization approach across teams in an organization
* Shift from delivery focus to value focus
* Engineering with observability as a service

Modern engineering can be a key enabler for business agility, allowing organizations to transform and move faster to compete in this rapidly changing business and technology landscape. It can transform new ideas into value quickly with the highest standards of quality.

Organizations transform because they’re looking for business agility, aligned with these four dimensions: speed, quality, cost, and culture.

The following are the main drivers for successful transformation through modern engineering practices:

* *Change-oriented pod teams delivering in small increments*: These pod teams are cross-functional, including all functions, to deliver the required user stories. The team must consist of product owners, business analysts, architects, required SMEs, and cloud SMEs.
* *Lifecycle management and configuration management*: To remove the code change paralysis, the team must have the proper lifecycle and configuration management in place with clear ownership of every aspect of the technology system.
* *DevSecOps*: The manual process is more error-prone. The team must unleash the benefits of automation with a shift-left approach.
* *Automated tests with production-like environments*: Teams must conduct automated tests, early and often, in production configuration.
* *Software-defined infrastructure and automated deployments*: With this practice, the team can move their changes faster to market.
* *Resilient, self-healing, cloud native systems*: You must prepare software for failures, learn from failures, and build resilient, self-healing cloud native systems.
* *Observability and automated operations*: Implement observability as a service with automated operation principles.

**Agile Software Development Methodologies**

Agile software development methodologies were created by leading software professionals based on real-time experience developing software. They address many of the limitations of traditional development methodologies. As a result, this is the de facto standard for cloud native application development.

Agile methodologies have certain values and follow certain principles of software development. These were referenced in the Agile Manifesto and the 12 principles of agile software, written by the thought leaders who created the agile software development approach.

There are various methodologies available for developing a cloud native application, and each of these adheres to agile principles and values.

**Hypothesis-Driven Development**

As an example, say you wake up one day in the morning and hear something that sounds like crying outside of your window. You might think it is a baby who is crying outside. Your hypothesis is that a baby is crying. Then you open the door and look outside. You know ahead of time that if you see a baby crying, you’re right, whereas if you see a playing baby, your hypothesis is wrong.

As another example, when you get into the office, you notice your end user isn’t using the sidebar link on the website. You might think the link is not in the proper place or it is not visible enough to the user. You decide to correct this by increasing the visibility of the link or placing it in the right place. Then you test the UI using A/B testing. You know ahead of time that if you see a statistically significant increase in clicks from the end user who see a visible link, that was the problem, whereas if you don’t see an increase, it wasn’t. When you run a test and see significant improvement, then you decide to roll it out to all the users.

In these two examples (the crying baby and the website link), you used the scientific method to test a hypothesis and create an effective solutions. It is thinking about the development of new ideas, products, and services—even organizational change—as a service of experiments to determine whether an expected outcome will be achieved. The process is iteration upon iteration until a desirable outcome is obtained or the idea is determined to be not viable. This is called *hypothesis-driven development*.

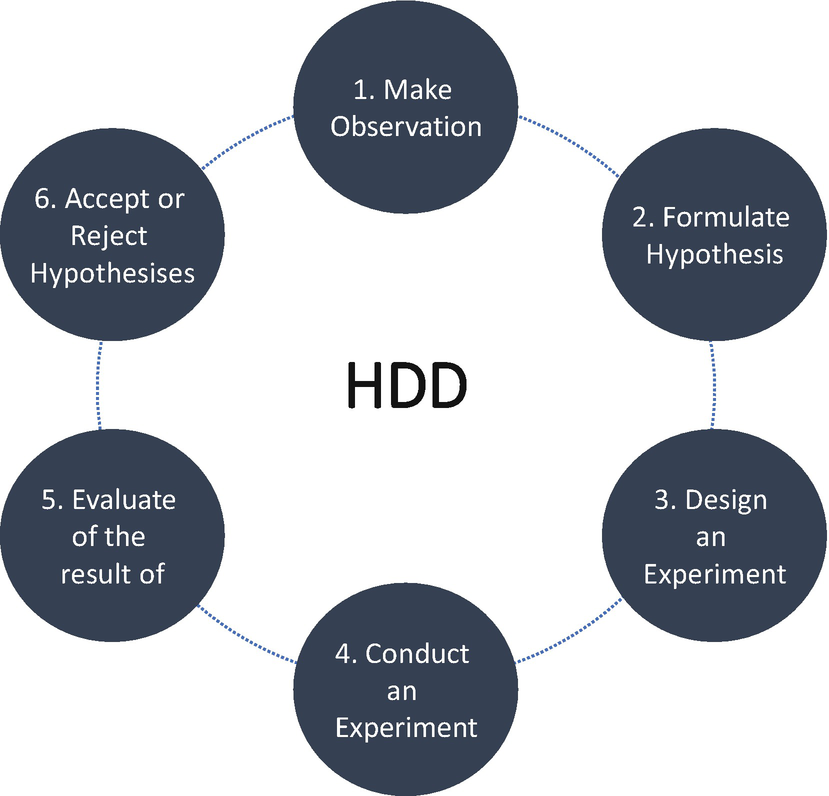
**Why Do You Need a Hypothesis?**

This is an approach that provides a structured way to consolidate ideas and build a hypothesis based on objective criteria. It’s also less costly for the system to test the prototype before actual implementation. Using this approach, you can implement the minimum viable product (MVP) model and identify what, how, and in which order testing should be done.

**Methodology Steps**

To facilitate a highly evolutionary approach, you need to use a hypothesis instead of requirements. Requirements are valuable when teams execute a well-known or understood phase of an initiative and can leverage well-understood practices to achieve the outcome, in other words, when something must be developed and delivered to the customer. The hypothesis is a provisional estimation that must be proven. If you disapprove a hypothesis, you need to pivot and create another set of hypotheses. When you use a hypothesis, you recognize customer expectations and needs and are constantly changing. To deliver what customers want at the speed that they demand, you must hypothesize and make data-based decisions. Experiment early and often, solicit feedback from customers about what works for them, and discard any features that provide little benefit to customers.

Figure [13-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Fig1) shows the scientific steps.



***Figure 13-1***

Hypothesis-driven development steps

Whether you are in the initial stage of your project or some other stage, there are always uncertain parts of an application, especially customer-facing applications, where you have ideas to further improve the existing product. To move forward, you’ll need to turn the ideas into structured hypotheses where they can be tested before production. Automation is a must for successful HDD. The process of the HDD scientific steps consists of the following:

* Test and track hypothesis experiments, by applying a feature toggle approach.
* When defining a hypothesis, you need to define data to validate the hypothesis, i.e., how much evidence you’ll need to make a decision.
* Test the hypothesis, and set up the test continually with automation to gather data for your decisions.
* Once you have a significant result, act on it; in other words, roll it out or roll it back. Note what worked and what didn’t, and keep running experiments.

Every user story in your project does not require HDD; identify use cases where you require HDD.

The success or failure of the hypothesis is always a learning opportunity, no matter what the outcome is. Even if you can’t prove the hypothesis, it provides valuable insight for another hypothesis.

Adopt a culture of hypothesis when you are developing a customer-facing and complex system. If a baseline test exists and the hypothesis asserts that an improvement to your service will be beneficial, you can conduct A/B testing to determine which option is best.

**Hypothesis Example**

Let’s say you are building a small ecommerce application and expecting more interaction from social media and certain links in other pages. The targeted users are very active on social media and the Internet, and you want to increase your traffic.

* *Hypothesis*: Providing an image of your product on social media will increase the number of targeted users who visit your ecommerce application and try the product. The changes will be measured by an increase in social media referral traffic to the link that is cited from the ecommerce site and other websites.
* *Outcome*: In user stories that deliver a new feature and tasks that require the posting images in social media or sent directly to the customers along with the specification and make the user aware of the product.

**Framing Hypothesis**

*“We believe that <this feature> will result in <the outcome>. We will know we have succeeded when <we see a measurable signal>.”*

* *We believe that <this feature>*: What functionality will you develop to test your hypothesis? By defining a test feature of the service that you are attempted to build, you identify the functionality and hypothesis you want to test.
* *Will result in <the outcome>*: What is the expected outcome of your experiment? What is the specific result to achieve by building the test capability?
* *We will know we have succeeded when <we see a measurable signal>*: What signals will indicate that the capability you have built is effective? What are key metrics you will measure to provide evidence that your experiment has succeeded and give us enough confidence to move to the next stage?
* *Hypothesis*: “We believe that if we write a blog on the feature of the product, people will want to buy it. We’ll know we have succeeded when xxx people visit and click the link.”

**Culture of Hypothesis**

Organizations must continuously practice and adopt a culture of hypothesis. You can practice in several ways by establishing metrics, establishing success and failure criteria, running multiple experiments continually by using automation, making data-based decisions, socializing across the organization by using various analytical methods, and considering a different model of experiments by using A/B testing.

Use metrics properly as these are essential to making data-based decisions, the metrics such as key performance indicators (KPIs) are often used to measure the hypothesis.

**Test-Driven Development**

First, you write a test , and then you write code to make the test pass. This approach to building software encourages good design, produces testable code, and keeps you away from over-engineering the system because of flawed assumptions. The problem you are facing in today’s world is poorly written services and failure to meet actual needs.

“TDD is a technique for improving the software’s internal quality, whereas acceptance TDD helps you keep your product’s external quality on track by giving it to the correct features and functionality.”

TDD is a way of developing software that encourages good design and is a disciplined process that helps you to avoid programming errors. TDD does so by making you write small, automated tests, which eventually build up an effective alarm system for protecting your code from regression.

The primary goal of TDD is to make the code cleaner, simpler, and bug-free. This is made possible because of the test-first approach adopted in TDD. Tests are likely to fail in the TDD process since the tests, specifically unit tests, are developed even before the code development. To pass the test, you have to develop and refactor the code. With this, you can avoid duplication of code since you’re writing a small amount of code at a time to satisfy the outcome of the tests.

**Why TDD?**

TDD will help shorten the programming feedback where tests are written before the functional code so that developers receive feedback on the quality of the code faster. TDD highly focuses on critical analysis and design because engineers cannot create the functional code without truly understanding what the desired result should be and how to test it. The process is also mandating that the code should not be written without the tests. This ensures higher-quality software.

The benefits of using TDD are as follows:

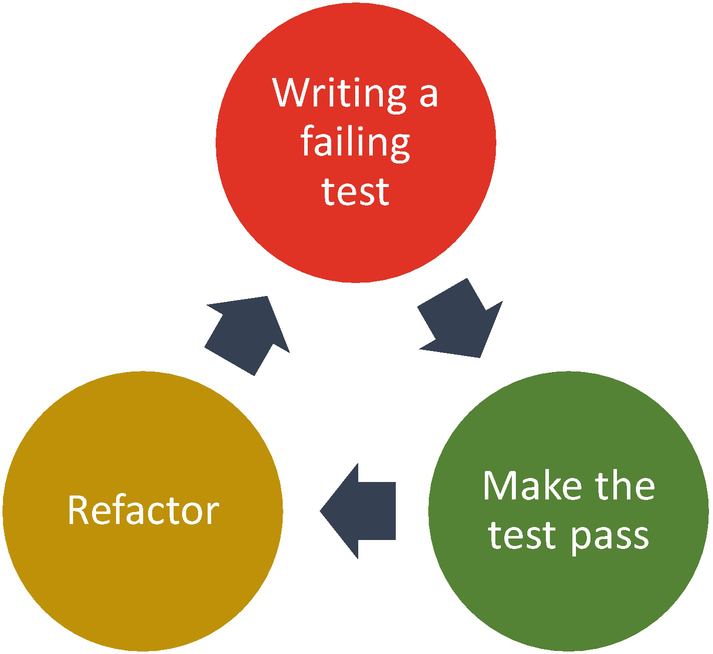
* TDD enables immediate feedback on the developed components.
* The turnaround time for the defect resolution is significantly shorter.
* TDD provides significant code coverage. TDD helps to make sure that every feature developed is wrapped with tests, resulting in increased test coverage.
* TDD helps to identify the problem in code quickly.

**TDD Cycle**

The TDD cycle as shown in Figure [13-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Fig2) is expressed as Red, Green, and Refactor and repeats for each unit of code.

* *Red*: Create a test that makes it fail.
* *Green*: Make the test pass by writing code.
* *Refactor*: Update the code to remove redundancy, and improve the design while confirming that the tests still pass after the update.

While using TDD, ensure 100 percent code coverage. The test-early principle helps you in detecting and fixing bugs early in the lifecycle of the product.



***Figure 13-2***

Test-driven development cycle

**Steps of TDD**

Kent Beck is the creator of Extreme Programming, a software development methodology that avoids rigid formal specifications for a collaborative and iterative design process. He sums up the five steps of TDD as follows:

1. 1.

Quickly add a test.

1. 2.

Run all tests and see the new one fail. Since there is no code yet to make the test pass, this test will fail.

1. 3.

Make a little change to pass the test as quickly as possible.

1. 4.

Run all tests and see them all succeed.

1. 5.

Refactor to remove duplication.

In the first step, you need to write a test. In the second step, you need to record a requirement as a test, and finally, you need to design the question and answers. The questions should be like the following:

* Does the method name reveal the intent?
* What are the parameters of the method?
* What is the outcome of this method?

The answers to these questions become the design decision that you express in code. In the third step, you always need to write the simplest possible code that makes the test pass. This allows you to keep your option open for evolutionary design.

Writing a failing test is a way of testing the test. If the tests all pass, it gives you feedback about your services that there are no known problems. By writing tests to expose a deficiency, that helps to identify the problem. Everything else is a best-painted picture. Ask yourself the following questions when you are writing a fail test:

* What is the responsibility of the system under test?
* What is there to observe?
* How is correctness defined?

You make your test pass with the simplest code. Is it too hard to pass the test? Then you drop back to changing your test and making it easier to pass.

TDD is a simple technique, as it has only a few steps to follow. However, in a real implementation, the steps are not that easy to follow since engineers need to be very disciplined. To get all the benefits of TDD, you should follow each step.

**Factors to Consider for TDD**

The following factors need to be considered while implementing TDD:

* Use the appropriate unit testing tool suitable for your project needs.
* Use the appropriate mock frameworks and code coverage tools.
* All the team members must agree on what level of testing occurs before integrating code to the source repository. You must restrict the tested code to check it in to the source repository.
* When the build breaks, what steps should be taken, and who will be involved?
* Use a proper naming convention for tests.

**Drawbacks of TDD**

TDD is good for cloud native applications, but it has a few drawbacks.

* It fits very well with unit test tools but does not scale with web-based GUI or data-driven development.
* Writing and maintaining an excessive number of tests costs time.
* For complex cases, the test cases are difficult to calculate.

**Behavior-Driven Development**

Behavior-driven development (BDD) is a process designed to aid the management and development teams by encouraging collaboration across roles to build and share an understanding of the problem to be solved and by working in rapid, small iterations to increase feedback and the flow of value. It improves the communication across the business, development, and support teams and ensures all development projects remain focused on delivering what the business needs while meeting all the requirements of the user.

BDD has evolved from the TDD. It brings techniques and principles from TDD and domain-driven design (DDD) to utilize features of these approaches and focus on delivering the prioritized business value and a behavior-based vocabulary. BDD is not replacing your agile process, but it enhances it. It is a set of plugins for your existing process that will make your team more able to deliver on the promise of agile.

BDD aids in system implementation from a stakeholder or product owner point of view through the use of a given-when-then-style of representing tests or acceptance test criteria associated with user stories. It offers guidance on organizing conversations between developers, testers, and domain specialists.

**How BDD Helps You to Solve Problems**

If you are not writing well-crafted and well-designed software, you’ll end up with unreliable software that’s hard to change and maintain, and if you don’t know what you are building and fail to understand what features the business needs, you’ll end up with a system that no one wants.

TDD, clean coding, and automated testing help you to guarantee a successful project, but what you are developing must also benefit its users and business stakeholders.

Applications should not be developed in a vacuum. The applications are part of the broader business strategy, and they need to align with the business goals of an organization. At the end of the day, the software solution you are developing needs to help users to use it effectively.

BDD helps you define the business problem that you want to solve, gives a business value to the organization, and helps to answer the specific question of how the problem will be solved. That is what behavior you expect, and more important it provides a ubiquitous language for all stakeholders.

TDD focuses on the technical architecture and the code, whereas BDD is focused on the business function and system behavior as outlined in the business story acceptance criteria. In both cases, the unit tests are written before writing the code.

**BDD Principles and Practices**

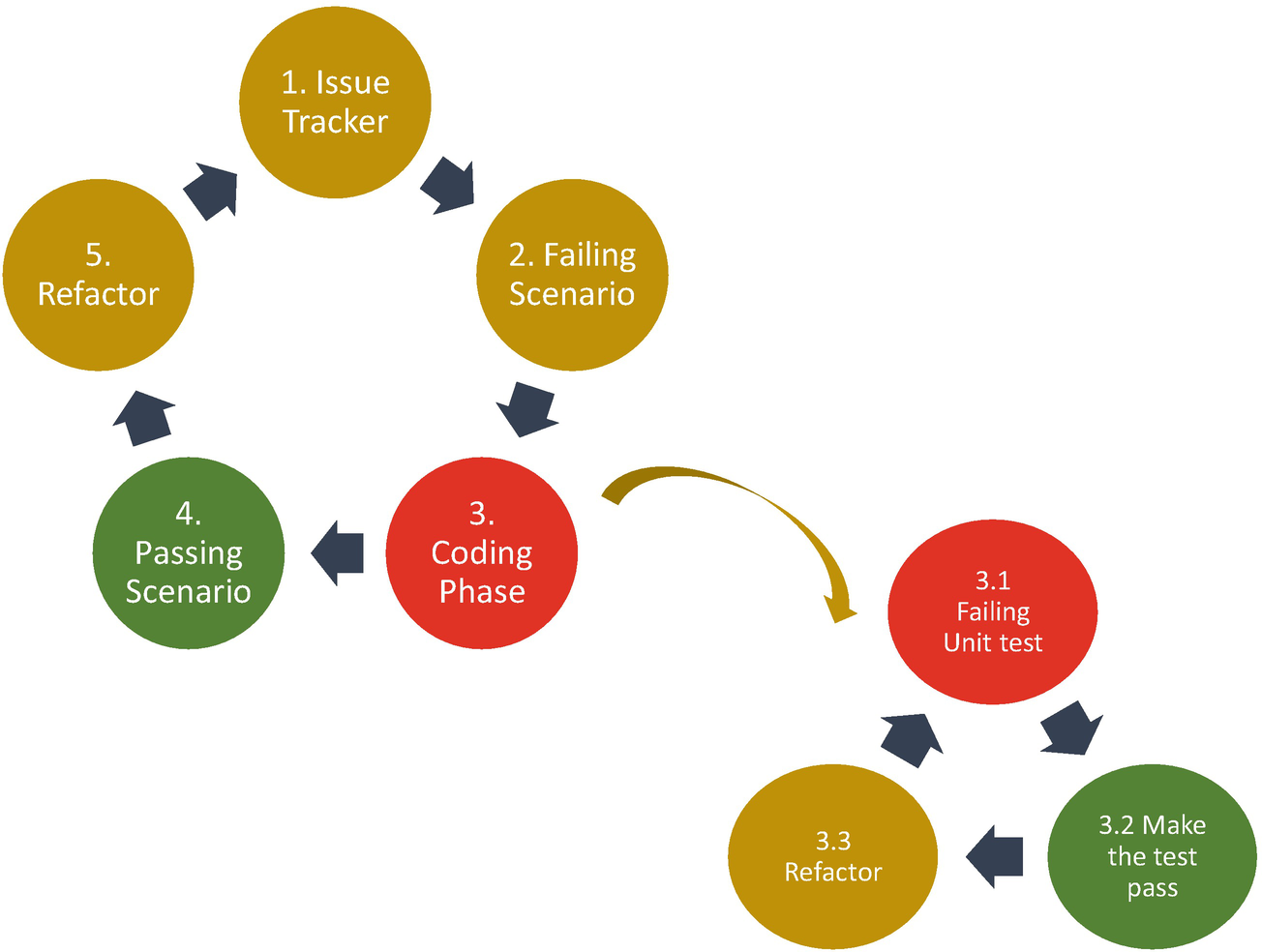
Here are some tips:

* *Focus on features that deliver business value*: Avoid heavy uplifting of the requirements specification. Rather, attempt to pinpoint all the requirements and engage customers or business stakeholders to progressively build a common understanding of what features they should create.
* *Work together to specify features*: BDD is a highly collaborative practice with various stakeholders who work together with end users to define and specify features. Team members draw on their individual experience and know-how.
* *Embrace uncertainty*: Rather than finalizing the requirements at the discovery of the project, BDD assumes that the requirements, or more precisely, their understanding of the requirements, will evolve and change throughout the lifecycle of the project.
* *Illustrate features with concrete examples*: BDD helps you to work together with the users and other stakeholders to define stories and scenarios of what users expect this feature to deliver, with a concrete example that illustrates the key outcome of the feature.
* *Don’t write automated tests; write an executable specification*: Write the executable specification as an automated test that illustrates and verifies how the system delivers a specific business requirement. These automated tests run as part of DevSecOps and run on each change.

**BDD Process**

As I mentioned, BDD is an advanced version of TDD, where “test first” agile testing practices are clubbed together by defining tests before, or as part of, specifying system behavior. This is a collaborative process that creates a shared understanding of the requirements between the business and development teams.

BDD is a collaborative approach used to test any application or service in a cloud native organization and supports a team-centric workflow. Figure [13-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Fig3) shows the process, which can be continued in parallel with the development phase.



***Figure 13-3***

Behavioral-driven development process

This approach uses real data in test along with the expected result. The data change is only in the feature files, not in the implementation code. With the use of a simple and natural language syntax, BDD breaks down complex requirements. BDD is more about collaboration and communication, and feature files can be written by anyone who has good knowledge of the domain requirements.

**BDD Specification**

BDD leverages a user story as the basic unit of functionality, and the acceptance criteria includes required components of a user story. It defines the scope of the user story’s behavior and provides a shared definition of done.

Each user story must have the following components:

* Name
* Narrative
* Acceptance criteria or scenarios

**Example**

* *Name*: Product returns from the ecommerce application go back into the stock.
* *Narrative*: To keep track of the stock, “as a distributor, I want to add items back to stock when they are returned.”
* *Scenarios*: Multiple scenarios are possible.
* *Scenario 1*: “**Refunded** stocks should be returned to stock; **given** a customer previously brought a pair of 34-inch blue jeans **and** I have currently 10 34-inch blue jeans in stock, when the customer returns the pair of jeans for a refund, then I should have 12 34-inch blue jeans in stock.”
* *Scenario 2*: “**Replaced** items should be returned to stock; **given** that a customer buys pair of blue jeans and I have 10 34-inch blue jeans in stock and 10 36-inch and 32-inch blue and black jeans in stock, when the customer returns the pair of jeans for a replacement of 32-inch black jeans, then I should have 8 32-inch black jeans in stock and 12 34-inch blue jeans, and there is no change in the rest of the stock.”

**Transition to BDD**

The following steps help you to transition to BDD from an existing process:

* Coach the business on BDD practices and the expectations from the business stakeholders on how to collaborate, the time required, and how to read the BDD reports.
* Prepare the functional team on how to write scenarios/behavior stories.
* Train teams to do the following:
  + Establish an understanding of BDD. The aim of the transition to BDD is to reduce the effort in the unit, regression, and functional testing.
  + Understand that development should be based on the scenarios or behaviors provided by the functional team.
* Educate development team and/or testing team to write BDD automation code for scenarios.
* Gradually implement BDD, starting with functional teams followed by the development team and then by the testing team.
* Use BDD to test main path scenarios and other test practices for testing border conditions.

**Benefits of BDD**

The following are the benefits of BDD:

* *Reduced waste*: BDD is all about focusing the development effort on discovering and delivering the features that will provide business value and avoiding those that don’t.
* *Collaboration*: BDD offers more precise guidance on organizing the conversion between all the required stakeholders.
* *Ubiquitous language*: All the stakeholders understand the same language from the requirements to deployment.
* *Business value*: BDD focuses on building features with demonstrable business value and not wasting effort on features of little value.
* *Easy to change*: BDD allows you to easily change and extend your system. Living documentation is generated from the executable specification that is understood by all the stakeholders.
* *Faster time to market*: Comprehensive automated tests speed up the release cycles. You use the acceptance test as a starting point and spend your time more productively and efficiently on exploratory tests.
* *Tests*: Reduce the time on regression tests to promote incremental development, and obtain coverage beyond unit testing.

**Drawbacks of BDD**

The drawback of BDD is more business involvement and collaboration. BDD is based on conversation and feedback across all the stakeholders including end users. If stakeholders are unwilling to participate in the conversation and collaborate, it will hard to get benefits from BDD.

* *Requirements*: BDD requirements are difficult and require a special skill to write requirements. Poorly written tests can lead to higher test maintenance costs.
* *It doesn’t work with the waterfall model*: BDD doesn’t work well with the waterfall model. In a waterfall model, each team works in a silo, and it’s difficult to collaborate.
* *Tools*: BDD uses tools that are not designed for use in large, complex projects that are difficult to customize.

**Feature-Driven Development**

Feature-driven development (FDD) is an iterative and incremental software development approach that combines several practices of developing client-valued features to deliver quality. It delivers frequent, tangible working results, via an accurate and meaningful progress, within a stipulated time. Features are an important part of FDD. A *feature* is generally a small client-valued piece of functionality expressed usually in the form of an action, or the result of an object. For example, validate the password of a user: the action is *validate*, the result is the *password*, and the object is the *user*.

**Why FDD?**

In software engineering, especially in a cloud native application, communication is taking place constantly at all levels of the software lifecycle. As the size of your system grows, the complexity of the system can become uncontrollable and untraceable.

FDD decomposes the entire problem domain into tiny problems, which can be solved in a small period of time. The decomposed problems are independent of each other and reduce the need for communication.

FDD splits the project into iterations so that the gap and time between analysis and test are reduced; this is a shift-left approach of errors.

FDD broadens the concept of quality, so you test not only the code but also things such as coding standards, measuring audits, and metrics of code.

**FDD Process**

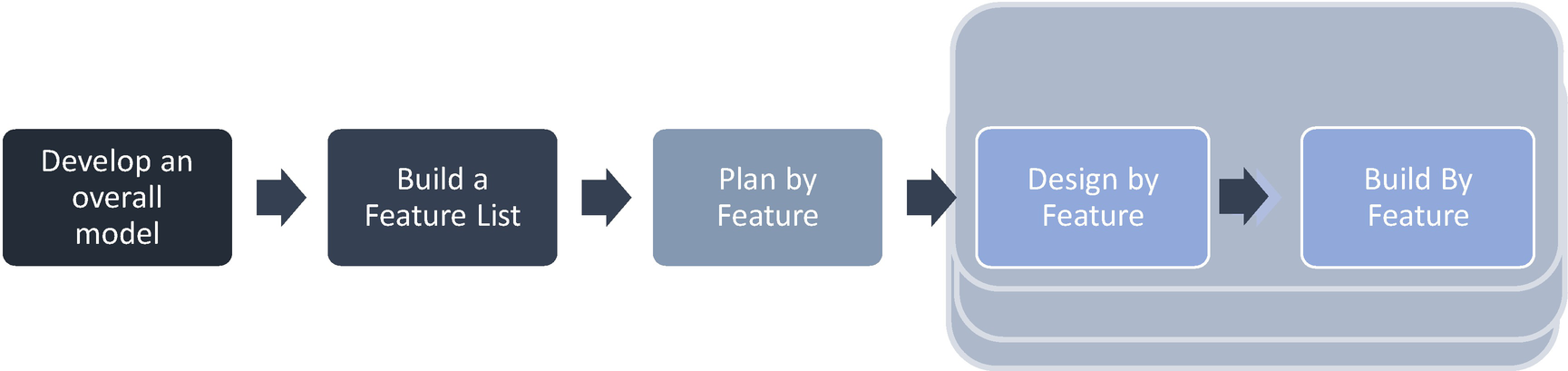
FDD defines milestones that mark the progress made on each feature. As shown in Figure [13-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Fig4), FDD consists of five processes.

1. 1.

*Develop an overall model*: Define a high-level domain for the system under development. The idea is for both domain and development members of the team to gain a good, shared understanding of relationships and interactions. In doing so, the whole team learns to communicate with each other. The object model developed in this step is breadth instead of depth. Depth is added iteratively in the lifecycle.

1. 2.

*Build a feature list*: Create a list of the state changes and interactions required, grouping them into feature sets and subject areas for planning. In this step, FDD defines a feature as a small, client-valued function expressed in the form of <action><result><object> (for example, “calculate the total of a sale”). FDD organizes its features into a three-level hierarchy called a *feature list*.



***Figure 13-4***

FDD process steps

1. 3.

*Plan by feature*: Agree on an initial schedule and assign responsibilities for each class and feature set. The planning team initially sequences the feature sets representing activities by relative business value. The feature set is also assigned to the scrum team, which is responsible for delivery. The risk and dependencies will be identified in this phase.

1. 4.

*Design by feature*: Create an initial sequence diagram that defines system interactions for each feature. The lead selects a group of features from one feature set that may be developed in a time-bound manner and forms a work package around those features that acts as a unit of integration with feature sets produced by the feature teams.

1. 5.

*Build by feature*: Develop and test the client-valued functionality to the point it can be deployed. After successful testing and code inspection, the completed feature will be promoted to the main build, where it will be verified and readied for release.

**Feature Specification**

A *feature* is a small slice of functionality producing a result of a value to a client, typically expressed in the following form:

*<action> the <result> [of | to |by| for| from|] a(n) <object>*

<object> is a person, place, or thing including roles, for example, “Calculate the total inventory,” “Authorize the sales transaction of a client,” “Provision credit card transaction,” etc.

Features are to FDD as user stories are to scrum. They are the primary source of requirements and the primary input into the team’s planning efforts.

Features should be small enough such that they can be designed, developed, and tested within a single iteration, that is, in less than two weeks. When your team feels a feature is taking longer, it should normally be broken down into smaller features.

**Feature Set**

Features are grouped into the feature set that represents the business activity or all the steps required to achieve a business objective. A feature set has the following form:

* *<action>-ing a(n) <object>*

Example include “making a transaction,” “adding a product to the catalog,” etc.

**Subject Area**

Feature sets are grouped into subject areas, subdomains of larger system, and are named with this form:

* *<object> management*
* Examples include “inventory management,” “customer management,” “product management,” etc.

**Benefits of FDD**

With the evolution of architecture, software development requires a slice of features to be available in production almost immediately. FDD helps you to define and develop a cloud native application easily and provides the following benefits:

* FDD provides a client-centric, model-driven approach for ensuring the frequent delivery of client-valued functionality.
* FDD concentrates on a small slice of design to enable the team to design, develop, and test.
* FDD is an effective approach for getting services to market faster.
* It offers improved communication across various stakeholders in a team.
* It works well with large-scale, long-term, or ongoing projects.

**Drawbacks of FDD**

While FDD offers a faster to market with slices by simplifying complex projects, there are a few drawbacks.

* FDD is not ideal for small projects.
* FDD places a high dependency on one role; this leader is required to coordinate across teams.
* FDD provides no written documentation to clients.
* It may not work well with the older systems.
* You might lose sight of your customers and instead only think in terms of features.

**Architecture in the Agile Methodology**

Architecture in agile projects can be built incrementally in sprints and can be moved into the design and develop phase or into another technical architecture sprint to implement. The first few sprints are dedicated to creating an overall architectural blueprint and later slices go into subsequent sprints for the detailed architecture and design.

Sprint 0 is used for the planning and preparation of architecture streams. The typical sprint duration for architecture is two to four weeks.

Perform the following tasks in architecture sprints:

* Conduct sprint planning to determine the technical requirements that can be addressed and how they should be addressed. The dependencies between various services need to be considered. The requirements should be documented in the backlog with priorities.
* For each architecture sprint, suggest creating a separate sprint backlog that is derived from the overall integrated product backlog.
* Perform the required testing before moving into the next sprint.
* Continually manage the scope of the project by evaluating and updating the product backlog.
* Transition the sprint deliverables to the application development to develop the code.

**Waterfall to Agile Transformation**

The transition from a waterfall methodology to agile should not be a sudden change or done in a day; it requires commitment and preparation. You need to carry out the following before transitioning to agile:

* *Culture of agile*: Commitment from management is a must; every project in an enterprise has to be executed to be aligned to the organization goals; hence, it is essential to commit sponsors or executives.
* *Training*: Without a skilled resource, it is not possible to transform, so training plays an important role in creating agile awareness and developing cultural shifts.
* *Coaching*: As you move from waterfall to agile, it is essential to have an expert who can work with the team and coach them.
* *Communication*: The Agile Manifesto states “individual and interaction” over “process and tools,” and hence communication plays a significant role for quicker turnaround time and a collaborative approach in agile.
* *Infrastructure*: Infrastructure plays a key role when you move from waterfall to agile. The infrastructure setup like source code, environments, automation, etc., should be available.
* *Metrics*: Agile adopts a minimalist approach toward metrics. It states you should measure everything required, but do not over-measure. These metrics help you to track the progress of the project and take corrective actions.
* *Estimation*: Estimation has to be precise since agile recommends projects to be executed at a sustainable pace.
* *Tools*: To get started with agile, it is recommended you use a set of tools that are required.

Figure [13-5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_13_Chapter.xhtml#Fig5) shows the steps you need to follow for transitioning to agile.



***Figure 13-5***

Waterfall to agile transition steps

* *Create a transition strategy*: The first step is to create a transition strategy for individual projects. Every project will have its transition strategy, which will differ from project to project.
* *Mobilize the transition*: Create a transition plan that will detail the timeline along with the milestones of each stage.
* *Implement releases*: Define the release structure.
* *Implement agile practices*: Start implementing a few basic agile practices so that the team can gradually adapt to the changes in the project execution process.
* *Continue with agile*: Introduce advanced agile practices.

**Summary**

As the complexity of software development in projects grows, the only way to maintain the viability of your build and ensure success is to have development practices grow with it. While the individual practices and processes of TDD, HDD, BDD, and FDD are all valuable in their own right, it is when they collaborate with each other that they provide a value to the cloud native journey.

In this chapter, I covered the software engineering principles for cloud native applications and explained how organizations can transform from traditional to intelligent engineering models. The way forward is product thinking, not project thinking, because you need to think about the end user’s behavior approaches, etc. I provided sample real-time examples of HDD and BDD scenarios.

Feature-driven development is a process for helping teams produce frequent, concrete, working results. It uses small blocks of functionality, called features. FDD organizes small functions into a business-related feature set. It focuses on getting a result every two to three weeks.

BDD is the process designed to aid in the management and delivery of cloud native software development projects by improving communication between the development team and business team and ensuring all related projects remain focused on delivery. The benefits of this approach are to help you to trace back to business objectives and develop a shared understanding of all the stakeholders with ubiquitous languages.

HDD follows the idea, hypothesis, design, experimentation, and scale steps. An iterative HDD process allows the engineering team to plan and conduct experiments; observe, analyze, and learn from results; and integrate the correct changes. This helps reduce uncertainty and improve knowledge. The engineering team must prioritize experimenting with the most uncertain aspects of the system.

TDD helps you to write tests and code. Next, it helps you to create optimized bug-free code and helps engineers to analyze and understand client requirements and request clarity. This helps you to test important new features in present-day scenarios.