CHAPTER 10

Modernize Monolithic Applications to Cloud Native

So far, I have explained various cloud native architectures such as microservices, event-driven, and serverless. These architecture concepts can be used for both greenfield and brownfield projects.

Business requirements can change, which is why old legacy systems in an enterprise may not support or meet the needs of business disruptions. There are various reasons that your enterprise must embrace modernization and go on a decoupling journey.

- Changing customer expectations and behavior
- Technology innovation
- New market entrants like unicorns (privately held startup company valued at over $1billion)
- Blurring industry boundaries
- Cost pressures

In this chapter, I will explain how you can modernize and decouple your enterprise's monolithic applications by using decoupling techniques.

In this chapter, I will answer the following questions:

- What does decoupling mean to you? Why do you need decoupling more than ever?
- What are the different approaches to follow your journey?
- What are the challenges you may face during the journey?

S. R. Goniwada, *Cloud Native Architecture and Design*, https://doi.org/10.1007/978-1-4842-7226-8_10

- How can you explore innovation while ensuring business continuity?
- How do you decide which systems require modernization?
- How will you plan the decoupling journey?
- What is the domain-driven design and approach?

What Is Decoupling?

In today's business environment and digital economy, organizations need to satisfy the existing customers and also need reach out to new customers across markets with more segments by expanding their digital offerings, without a comparable extension in IT and or market budgets.

Digital decoupling is the combination of strategy, approach, tools, and techniques to address speed to market at scale with designing for cloud native and when designing for a customer in an organization's IT estate burdened with years of technical debt. It is the concerted approach to exploit cloud native technologies to break down monolithic legacy IT, address technical debt, and transform to an ecosystem where IT changes are negligible.

Decoupling enables large enterprises to reassert competitive advantages against incubators or unicorns.

Decoupling involves replacing the technicalities of the IT system by keeping the business functionality to support revenue growth and add the greatest value to the customers. This way, your enterprises can respond to market forces and technological innovation while maintaining cost levels.

When decoupling at scale, this leads to @Scale IT, a scalable, flexible, and resilient architecture that gives your organization the agility to innovate at scale, streamline the IT estate and retire unused systems, and rationalize the portfolio to a singular function across the landscape. This helps your organization to compete with the unicorn companies on equal terms.

Decoupling embraces the use of cloud native architecture and software engineering methodologies to build new systems that execute on top of legacy systems.

@Scale IT is the emergence of cloud and cloud native technologies, various channels, artificial and machine learning with observability, and modernized software engineering with agility and AI-driven development. The more adaptive event-based architecture is called @Scale IT.

Technical Debt

> *"The price companies pay for short-term technological fixes hinders their ability to innovate and adapt in the digital age. One strategy to combat technical debt? Digital decoupling." —Adam* Burden*,* Edwin *Van der Ouderaa, Ramnath Venkataraman, Tomas Nystrom, and* Prashant P. *Shukla*

IT is not new, and the systems in your enterprise are probably not new either. As the business expands, the systems in your IT department become legacy by the nature of the fact that the environment around them, such as people, process, and technologies, progress while the systems remain relatively static.

Organizations face intense pressure to meet the business disruptions, competition from unicorns, and customer expectations. To support this, enterprises are adding more features into the existing legacy systems, which results in technical debt that leads to more operation overhead. The decisions that resulted in technical debt were likely not wrong at that time; they were made to enable the business. However, if not properly paid attention to, the debt will continue to grow at an alarming rate. In the end, you need to spend more money on maintaining an application than on innovation and new technologies. Over time, the enterprise faces a lot of challenges when updating these systems. This becomes devastating for IT teams, and digital transformation becomes more difficult.

How Are Technical Debts Accumulated?

As explained, technical debt is a normal result of software engineering. Some debt occurs for good reasons, and some occurs unintentionally.

The first type of technical debt occurs when an enterprise IT team makes an informed decision to generate some technical debt and is fully aware of the consequences due to various reasons. The reasons can be to meet the delivery timeline, meet a resource crunch, create business functionality in production, etc. These decisions can accumulate quickly over time.

The second type is unplanned technical debt that arises due to poor practices, inexperienced teams, no review and checks, poor understanding, etc. This poor management, poor communication, or misalignment can accumulate over time.

The third type is business and technology change. These debts are unavoidable due to business disruption and better technology and solutions being available. It

typically accumulates by adding more features to the existing systems to support the new business without changing the technology.

In a nutshell, the technical debt stems from everyone's carelessness, bad decisions, and other reasons. Figure 10-1 shows Martin Fowler's technical debt quadrant. I have modified the quadrant to suit present-day software engineering.

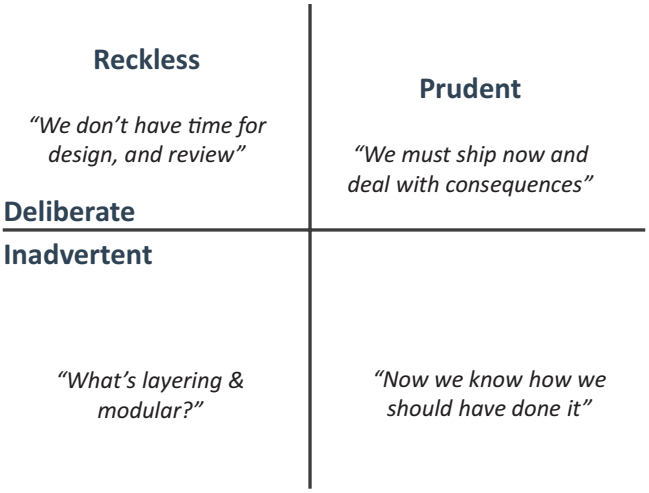


Figure 10-1. *Technical debt quadrant*

How Is Technical Debt Impacting Your Enterprise?

Technical debt makes your enterprise uncompetitive against peers or unicorns; it makes it more difficult to add new business value to the software and makes fixing problems more challenging. This will reduce the overall asset value and create greater risk in managing the portfolio of assets. These are critical inflection points where constraints move beyond IT to threaten core mission, business, and operational programs. They occur when accumulated technical debt causes these critical systems to either chronically break down, decline, or become so inadequate, sluggish, or inflexible that your organization is forced to halt or significantly slow down investments on innovative new cloud native systems until it consolidates, replaces, or rearchitects existing systems into cloud native.

The leading causes of these events are legacy systems, lack of resources for maintenance, inability to add new features and integrate across enterprises, including poor maintenance and inadequate investments.

How to Decide on Decoupling?

As explained, technical debt can arise across enterprises irrespective of decision levels; everyone in an organization is responsible for technical debt.

Technical debt is a metaphor that, just like in finance debt, incurs interest payments. This means technical debt makes your enterprise's IT more expensive to maintain than it has to be. This is a direct impact on your business. The following section will help you to measure technical debt in your organization to decide on a decoupling journey of a system. I call this method the *decoupling model*.

Decoupling Model

Technical debt doesn't help decision-making if we can't do an analysis. Once we quantify technical debt, we can make an analytical comparison.

Legacy cost: This is the largest debt in any organization, and it is easy to measure. It includes the cost to remediate and maintain in-house legacy and vendor products. Like financial debt, you make measurable progress in debt reduction by paying down the principal.

Variable cost: These are the costs related to staffing, reviews, tools, delays, and duplicating systems that must be maintained. By not reducing the legacy cost, the variable cost is unavoidable and incurred.

Maintenance cost: The legacy systems become fragile and vulnerable. Outages, breaches, and data corruption occur, leading to significant cost for the maintenance of replacing software, hardware, etc.

You need to consider many variables to determine the effect technical debt has on computation. Some of the variables include complexities, lines of code, maintainability index, Halstead complexity measures, etc.

Technical Debt Ration (TDR) = (Remediation Cost/Development Cost) *100%

Remediation cost is a ratio of the cost to fix a software system, and development cost is the cost of development.

Always keep the TDR below 5 percent. If the TDR is above 5 percent, then it's time for you to take action to decouple the legacy system.

Remediation cost (RC) is the maintenance cost of a system. The RC is directly proportional to the cyclomatic complexity of your code.

RC = k(cyclomatic complexity)

Cyclomatic complexity is a metric used to indicate the complexity of a program. You can get cyclomatic complexity from review tools like SonarQube or CAST Software, and k is the constant.

Development cost (DC) is a variable cost for writing some lines of code. For example, if a file has 100 lines of code (LOC) and the average time to fix is 20 minutes to write one line of code, the cost per line of code (CPL) is 20 minutes.

DC = 25/line * 100 lines = 2500 minutes =2500/60 = 41.66 hours

To calculate whether your application requires a decoupling, use this formula:

LOC = 25,000
RC = 735 hours
DC = 0.42/line. DC = 0.42*25000 = 10,416 hours
TDR = (RC/DC) *100%
= (735/10416)*100% = 7.05%

Your application TDR is 7.05 percent. In this example, this application requires you to undergo a decoupling method to move into the cloud native application.

Decoupling

Based on research from a leading consulting company, as many as 81 percent of organizations indicate that they would like to replace their legacy core systems with a cloud native architecture.

As mentioned, *decoupling* is the process of decoupling monolithic legacy applications by using new technologies, development methodologies, and migration methods to build new systems that execute on top of legacy systems. For example, by using application programming interfaces (APIs), agility, automation, and cloud native, you can gradually decouple core systems, migrating critical functionality and data to new platforms.

Decoupling is required in present-day architecture because of the following:

- *Changing customer expectations*: You need to connect more with the customer with meaningful customer relationships and provide a great user experience based on individual tastes to make a customer's life easier.
- *Technology innovation*: Your business must be accessed by any user on preferred devices without interruption, and your system should be able to provide real-time analytics based on real-time feedback.
- *New market entrants*: The rise of unicorns without any legacy baggage shifts the market share.
- *Blurring industry boundaries*: Your system must be able to adjust to real-time demands from the customers by rearranging the value chains and providing real-time analysis on pre- and post-sales.
- *Cost pressure*: Organizations are under immense pressure to deliver a higher level of services at lower cost and to remove legacy infrastructure, reengineering processes, and rationalizing workforces.

As I mentioned, 81 percent of organizations want to move to cloud native, especially after the COVID-19 pandemic, by removing the legacy applications, but organizations are taking too long to decouple legacy services.

Legacy services are the main drag force to innovation and digital transformation; however, as shown in Figure 10-2, making changes to legacy services is difficult because:

- We are dealing with tiered systems, designed to operate as a whole.
- Individual components are highly coupled and interdependent.
- Making any small change inevitably causes a ripple effect that must be mitigated or adjusted for.
- The change will take a long time and be expensive.
- It's hard to know where to start and exactly what to change. See Figure 10-2.

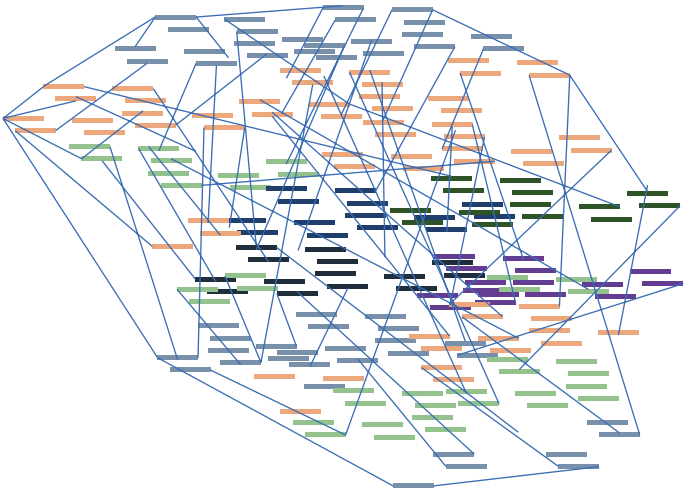


Figure 10-2. *Monolithic legacy application in an IT estate*

If you want to build cloud native technologies around a monolithic, it adds more complexities. With every addition into the monolithic system, the cost of testing, enhancement, and operation will increase.

As shown in Figure 10-3, legacy systems are typically dominated with a large and highly complex monolithic business layer. The legacy core is a tightly coupled monolithic architecture that acts as a brake on innovation, agility, and cloud native.

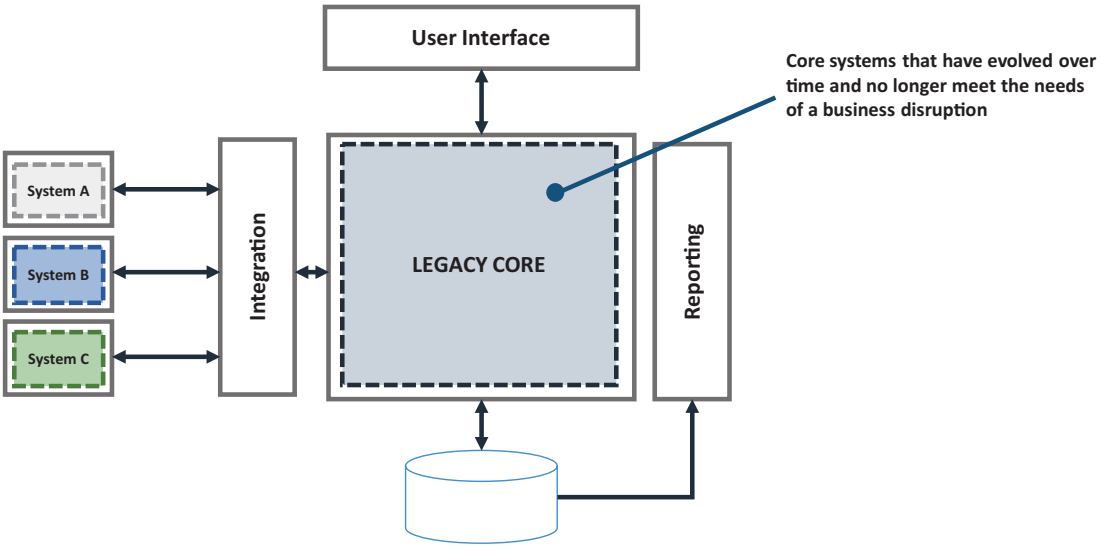


Figure 10-3. *Typical monolithic system*

A tightly coupled architecture increases delivery timelines, operations, and risks. The impact assessment on change requests, the lengthy testing cycle to test the entire legacy core for small change, and the complex code all add to the uncertainty.

Figure 10-4 indicates how organizations can change to cloud native systems. Usually, the organization puts in five to ten years of transformation, which leads to system flaws, complexities, and inefficiencies of a system. If you do not adopt decoupling early, then it will be too late to come out of the mess and you might lose the customer base.

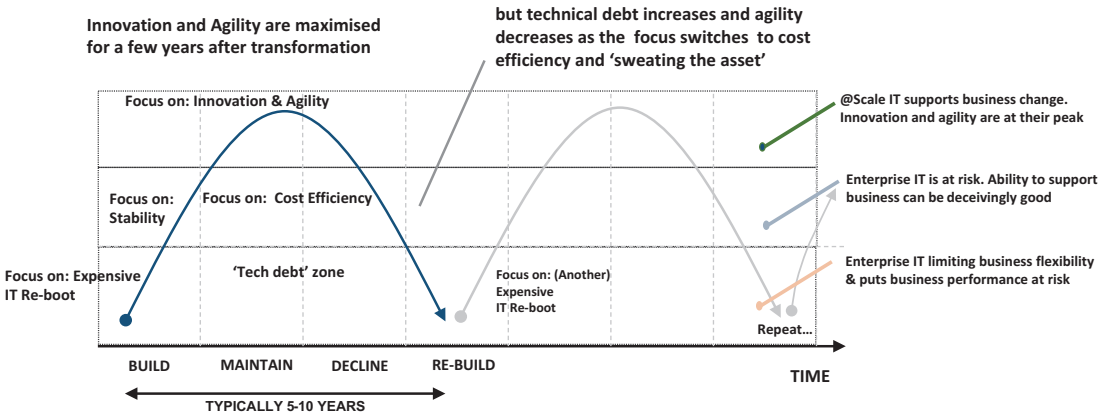


Figure 10-4. *Organization's approach on digital transformation without decoupling with cloud native*

As I mentioned, the digital economy has changed the competitive landscape, allowing new entrants to seriously challenge incumbents and change the market overnight. Figure 10-5 provides a high-level comparison of decoupling across unicorns, early adopters, and laggards. This is a lesson for you to adopt to cloud native early in the lifecycle by decoupling the legacy applications. This graph should open your eyes to the importance of decoupling.

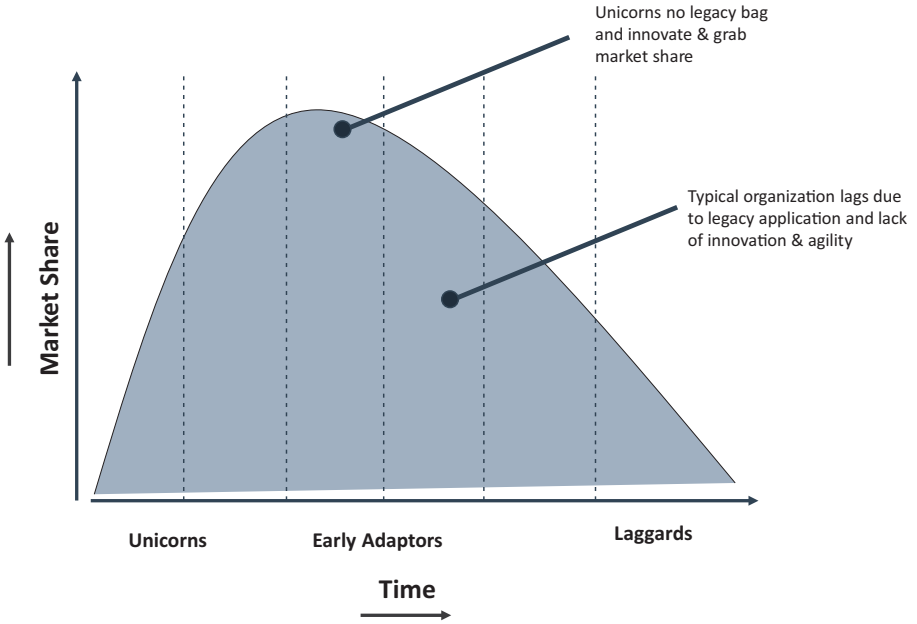


Figure 10-5. *Comparison between unicorns and traditional organizations*

Decoupling Approach

In the decoupling approach, the legacy core as shown in Figure 10-3 is evolved and provides the business transactions to the customer. But due to disruption in the business and changes in customer expectations, today organizations do not meet the needs anymore. A user interface is heavily relying on the legacy core as shown in Figure 10-3 with a tightly coupled architecture. An entire stack of the system is deployed on-premises on a virtual machine or bare metal.

As shown in Figure 10-6, the journey to a decoupled architecture starts with the implementation of automation for the existing legacy core. In parallel, the organization builds the cloud native services and deploys them on-premises and in the cloud, respectively, to exchange data. All the "Specific New" applications are converted into cloud native services with event-driven architecture and exposed as APIs to the web and mobile interfaces with real-time interaction. Those services are decoupled from the database by adopting the polyglot persistence principles and syncing them with the legacy core database for other transactions.

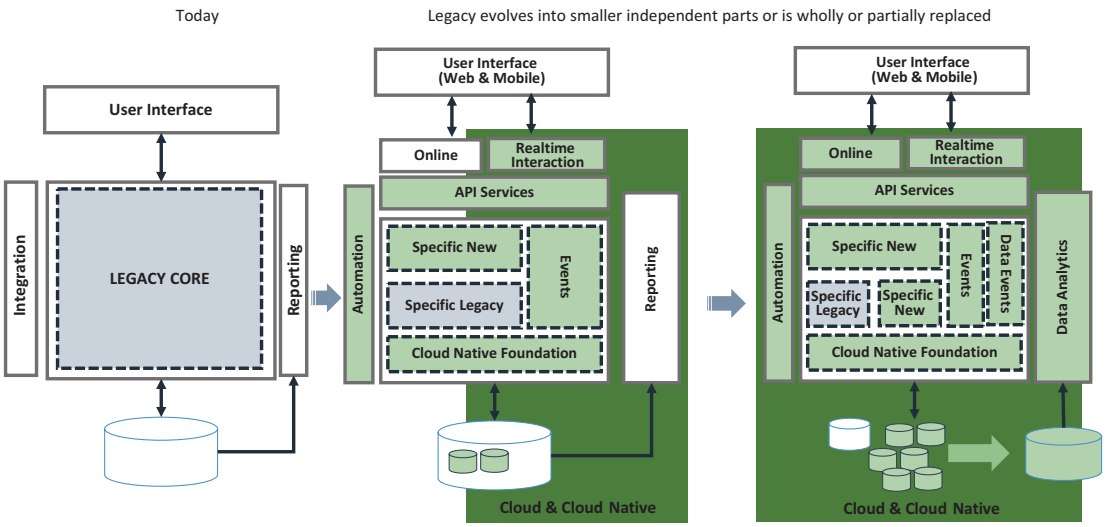


Figure 10-6. *Decoupling approach*

Finally, incrementally, the entire legacy core is decoupled into a cloud native system with polyglot persistence, and data is replicated to the data lake or data mesh for analytic purposes.

Following the decoupling approach, the organization introduces integrated monitoring or observability, real-time data lake integration, and systems of intelligence for smart interaction.

The end goal is to transform the monolithic legacy core system into a cloud native platform to fully unleash its value while eliminating current constraints. This requires understanding of future roadmaps, the business, and customer behavior.

Large-scale application modernization projects usually encounter overruns and failure. Adopting agility, automation, strangulation, iterative development, and incremental delivery can reduce risks while accelerating time to value.

The decoupling and continuous modernization for your organization is important because it allows enterprise IT to be more responsive to business changes and is easy to prioritize based on the demand. In addition, it helps to guard against the accumulation of additional technical debt through regular upgrades and follows the easy to create and easy destroy principle.

As you move toward @Scale IT, your enterprise can evolve toward a true service-based IT architecture that maximizes agility. This provides rules-based decision-making across the organization.

Decoupling Plan

You must follow the iterative MVP approach shown in Figure 10-7, not the big-bang approach. If you follow the big-bang approach, the decoupling projects will fail.

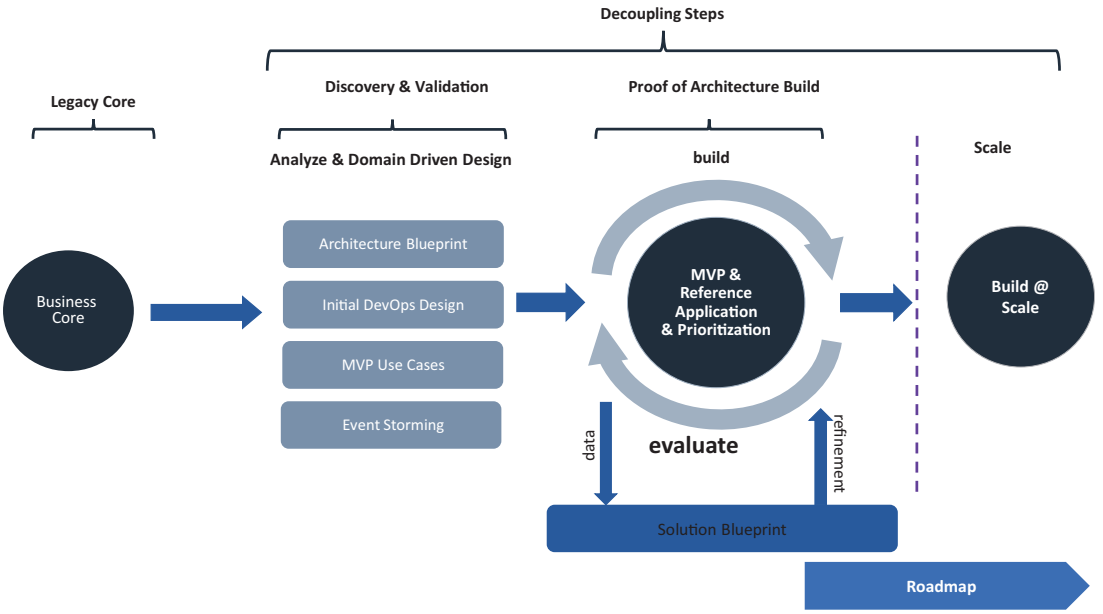


Figure 10-7. *Decoupling approach and plan*

Follow these steps for the decoupling approach:

1. Identify a system in your enterprise to start decoupling.
2. Create an architecture blueprint.
3. Conduct a design thinking session and domain-driven design with an event storming exercise to identify the microservices.
4. Initiate DevSecOps.
5. Identity an MVP use case.
6. Create a proof of concept (POC) and reference architecture.
7. Deploy the POC along with the legacy core and evaluate.
8. Once it is successful, create a solution blueprint.
9. Finally, go with scale to decouple the entire legacy.

Decoupling Principles

Use the following principles during the decoupling process:

- *Layering*: Apply layering to isolate parts of the core system.
- *Appropriate fragmentation*: Fragment capabilities to remove conflicts of interest and increase agility.
- *Simplification*: Simplify systems and keep differentiated logic separate from commoditized logic.
- *Differentiated services*: Build out the systems of differentiation to support reuse, automation, data analytics, and agility.
- *Cloud*: Leverage cloud native capabilities to quickly adapt and build services.
- *Intelligence built-in*: Build systems with AI and ML in them to enable smart interaction.
- *Event-driven*: Build an application that supports asynchronous events.
- *Real-time data*: Build data lakes or data meshes with real-time eventing capabilities to support the services.
- *Prediction-based model*: Add prediction across the application for self-healing and infrastructure prediction.
- *Observability*: Build a system with observability as a service.

Decoupling Business Case

When you consider decoupling the legacy core, you may be required to create a business case for your leadership before initiating @Scale IT.

You need to conduct the as-is assessment of the existing system and do a decoupled architecture assessment to compare the costs in order to determine the value of the decoupling.

For the as-is assessment, consider the following:

Overall, as-is cost = *Infrastructure cost + platform license cost (app server, DB etc.) + Resource cost (people) + Maintenance cost + deployment cost+ opportunity loss (delay in future loss, sales impacted etc.)*

For the target state assessment, consider the following:

Overall target cost = *Infrastructure cost + License cost + People cost + Refactoring cost + Maintenance cost + Benefits*

Cost-Benefit Analysis = Overall as-is cost ~Overall target cost

Decoupling Strategies

The transition journey from the legacy core to a cloud native architecture requires an incremental approach to decouple the legacy core and integrate it back to the legacy core. The transition journey begins with a strategy. The following are the decoupling strategies you need to adopt for your journey:

- *Service decomposition strategy and roadmap*: This strategy identifies objectives, business context, and priorities. Assess the current architecture; you can refer to Chapter 11 for an assessment approach to incrementally refactor the legacy core into cloud native. Develop a high-level roadmap based on the business value impact. Identify the use of any relevant standards and adherence to cloud native governance and compliance requirements.
- *Decoupled architecture and integration planning*: This strategy defines the integration architecture for routing requests between the new services and the legacy core as well as enables the service to access data and functionality from the legacy core.
- *DevSecOps strategy*: This strategy defines the technical deployment infrastructure and delivery model required to build continuous deployment and defines infrastructure as code for automating the infrastructure service to deploy cloud native applications.
- *IT operating model*: This strategy defines an integrated intelligent IT operating model to organize around systems and value generation. Adopt a Intelligent Operation as explained in Chapter 18 to enable

a faster time to market for decoupled services. Here you analyze the organizational impact based on the recommended changes to people, process, and technology for a decoupling journey.

- *Value case*: This strategy develops the business case to determine how a decoupled architecture can be implemented to deliver greater value to meet business disruption.

Domain-Driven Design

Domain-driven design (DDD) is an approach for developing software for complex needs. In this method, the implementation is a constantly evolving model to match the core business. The concept was first introduced by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

Before getting into DDD, let's first understand why you need DDD and what are the difficulties of creating and maintaining a software system. Brian Foote and Joseph Yoder have defined a pattern called *big ball of mud* (BBoM). The definition of BBoM is "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle."

A BBoM system appears to have no distinguishable architecture. The issue with allowing software to dissolve into a BBoM becomes apparent when routine changes in workflow and small feature enhancements become a challenge to implement due to the difficulties in reading and understanding the existing codebase.

Eric Evans describes such systems as containing "code that does something useful, but without explaining how." As shown in Figure 10-8, this is one of the main reasons systems become complex and difficult to manage, mixing the domain with technical complexities.

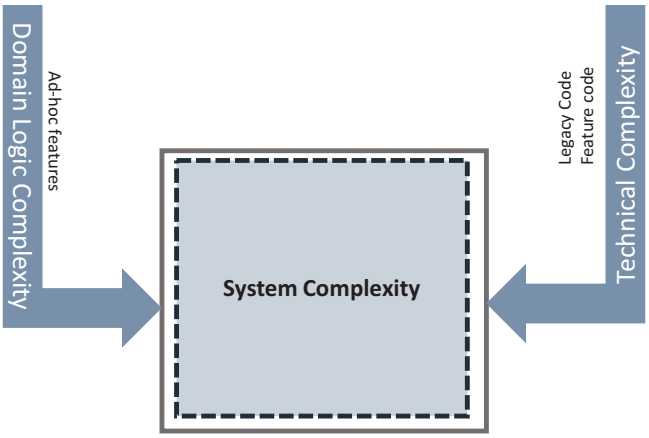


Figure 10-8. *Complexity in software*

A lack of understanding of the domain, the ad hoc introduction of code, and improper management in the source code repository makes the codebase difficult to interpret and maintain because translation between the design model and the development model can be costly and erroneous. Let me explain in a real-time example how improper management of code becomes very costly.

Continuing to persist with an architectural spaghetti-like pattern can lead to a sluggish pace of feature enhancement. When newer versions of the project are released, there can be mismanagement of the codebase. Over time, this problem grows and becomes unmanageable.

I was working as an architect with one client that had a monolithic tightly coupled web-based architecture. About 100+ software engineers were working on this huge complex platform, and they spent nearly two to three weeks identifying the right branch in the source for building and deploying changes to the production. When I conducted an analysis, I found nearly 2,000 branches in an SCM, and no one knew why they were created. This illustrates how code management can become complex over time if you do not manage it properly.

How Does Domain-Driven Design Manage Complexity?

DDD deals with the challenges of understanding a problem domain and creating a maintainable solution that is useful to solve problems within it. DDD uses strategic and tactical design principles to define a domain-based design.

What Is a Domain?

A *domain* is the knowledge and activity around which the application logic resolves; in other words, it is the business logic that is the core of the system.

There are three types of domains.

- *Core domain*: The core domains are the most strategic domains for the business at the enterprise level and program level. This is software that you build and is a differentiator.
- *Supporting domain*: The supporting domains are required by the core domain and are either built or are commercial off-the-shelf software (COTS) or SaaS.
- *Generic domain*: Generic domains are likely implemented by selecting commercial software/services or open source software, e.g., identity and access management.

Goals of Domain-Driven Design

The following are the goals of DDD:

- Build software that has a complex business process (business domain) while the knowledge is limited.
- Identify a bounded context and its patterns of interaction to enable independent deployment teams.
- Separate the business model (business logic) from the implementation details.
- Collaborate between technical experts and domain experts to implement a solution that works seamlessly.
- Create a ubiquitous language for each bounded context to use among business architecture and software engineers throughout all phases of development.

Domain-Driven Design Model

DDD is about distilling the legacy core into cloud native services. Figure 10-9 illustrates the various exercises needed to identify a service. DDD is distilled into strategic and tactical DDD.

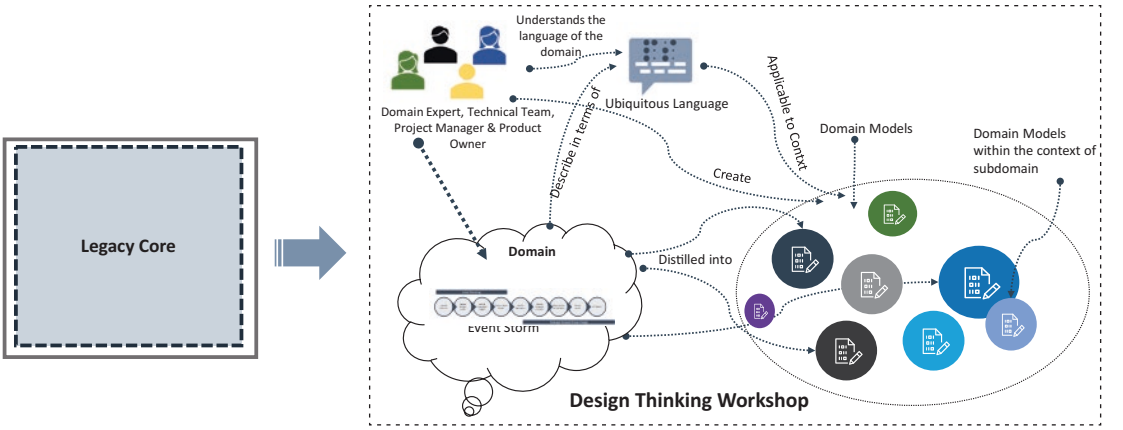


Figure 10-9. *DDD model workshop in a single diagram*

Strategic DDD

Strategic DDD distills the problem domain and shapes the architecture of an application. The technical team, product owner, and domain experts use the design thinking method to distill a large and legacy problem domain into microservices. DDD emphasizes the need to focus effort on the microservices as these hold the most value and the way forward.

Holding a design thinking workshop on the core domain helps the team to understand the domain in the legacy core and how important this domain is in the business. It will enable the software engineering team to identify and invest its time in the important parts of the system.

The outcome of DDD is to identify a well-defined cloud native architecture solution with microservices as its core and to identify the domain stories without changing the core domain business logic and rules.

The cloud native services are built through a collaboration of domain experts, product teams, and technical teams. Communication is achieved using an ever-evolving shared language known as the *ubiquitous language* to connect cloud native services

efficiently and effectively to a conceptual domain model. The cloud native services are bound to the domain model by using the same terms of the ubiquitous language for its structure and class model.

Cloud native services sit within a bounded context, which defines the applicability of the services and ensures that their integrity is retained. Larger services can be split into appropriate services and defined within a separate bounded context where ambiguity in terminology exists or where multiple teams are participating in a design thinking (DT) workshop to further reduce complexity. Bounded context is used to form a protective boundary around services that helps to prevent software from evolving into a BBoM. This is achieved by allowing the different models of the overall solution to evolve within well-defined business contexts without having a negative, rippling impact on other parts of the service.

Tactical DDD

Tactical DDD is a collection of various cloud native patterns that help to create effective services for complex bounded contexts. Many patterns are explained in Chapter 4. You can use these patterns appropriately for each service instead of adopting them randomly.

Guiding Principles of DDD

There are practices and guiding principles that are key to the success of DDD.

- *Focusing on the core domain*: The core domain is the area of your system where most of the business logic resides. The behavior and functioning of your system depend on the core domain.
- *Collaboration across a team of experts*: This stresses the importance of DT workshops that allow brainstorming with the technical team, domain experts, and product owners. Without collaboration across teams, much of the knowledge sharing will not be able to take place.
- *Use domain terminology in the code*: DDD treats analysis and code as one, which means the technical code model is bound to the analysis model through the shared ubiquitous language. Use domain terminology in code to reflect the business language.

- *Communication*: The single most important facet of DDD is the creation of the ubiquitous language. Without a shared language, collaboration across teams would not be effective. It is the collaboration and construction of a ubiquitous language that makes DDD is more effective. It enables a greater understanding of the problem domain and more effective communication.
- *Continuous evolving*: Without the synergy between the code and domain language, you will end up with a codebase that is hard to modify, resulting in a BBoM.

How does it help you?

- DDD provides a logical approach for identifying subdomains to convert a legacy core system to multiple relatively independent cloud native services.
- DDD allows you to identify subdomains for specialized treatment based on specific needs.
- DDD enables the identification of core, supporting, and generic domains, with each domain capable of being deployed independently of others.
- The DDD process is a powerful tool for business and delivery teams to be on the same page regarding core code.
- The business process shares a common vision of what is important to the business.

Event Storming

Event storming is a design thinking workshop for the collaborative exploration of complex business domains and a modeling approach to domain-driven design. It was created by Alberto Brandolini in 2012 as a quick alternative to Unified Modeling Language (UML).

As shown in Figure 10-10, event storming in DDD consists of a four-step approach.

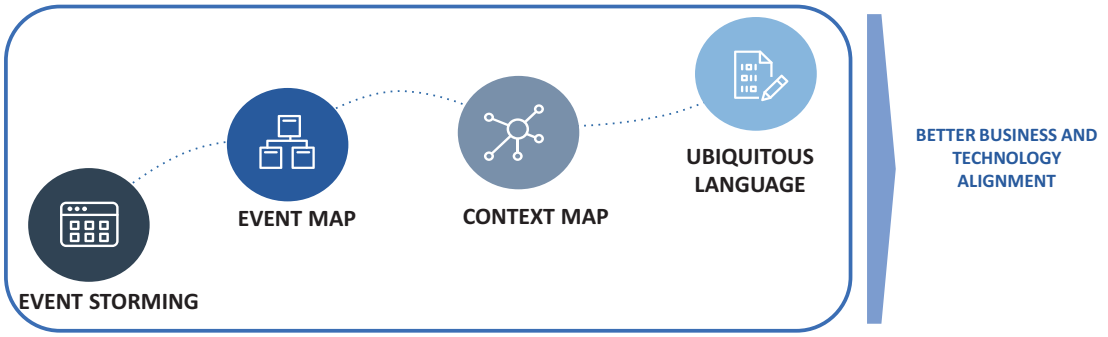


Figure 10-10. *Event storming in a DDD*

Event storming: This consists of design-level modeling and focuses on domain events and business process.

Event map: Business processes are documented using events, commands actors, and external systems.

Context map: This is a visualization of boundaries, dependencies, and communication paths between cloud native services teams.

Ubiquitous language: This is a clearly defined language used for all discussion between product teams, architecture, and engineering. It is a model that acts as a universal language. This is needed for understanding and communicating concepts in the domain in an unambiguous manner and improves collaboration with domain experts in order for everyone to be more creative and valuable. This must be expressed in the domain model to unite participants and eliminate inaccuracies and contradiction.

Key Roles in an Event Storming Workshop

These are the key roles:

Domain experts: These are the business representatives who understand the product vision and the target state's business process.

Architect: The architect and designer will be building the final solution.

Facilitator and DDD practitioner: This person facilities the event storming workshop. The project manager can act as a facilitator.

Product owner: This person prepares personas, stories, and event storming output with the integrated backlog for the implementation team.

UX designer: This persons is fully engaged in the modeling activity to push the process toward innovation and customer-centric thinking.

Data analyst: Many solutions in a cloud native system use data. Engaging a data analyst during design will ensure data implications are thought through at the beginning.

Event Storming Exercise

As shown in Figure 10-11, the event storm is a nine-step design thinking workshop that brings together domain experts, technology team, product owner, and project manager to model and understand the business process. It is *not* a technical design session nor an exploration of the current state of the architecture. The goal is to understand the ideal business process, *not* the current or future technical implementation.

Event storming is a:

- Conversation starter
- The evolving model of problem and solution
- Tool to gather requirements and build an event-based view of an event process
- A visual reference to view problem areas and possible solution paths

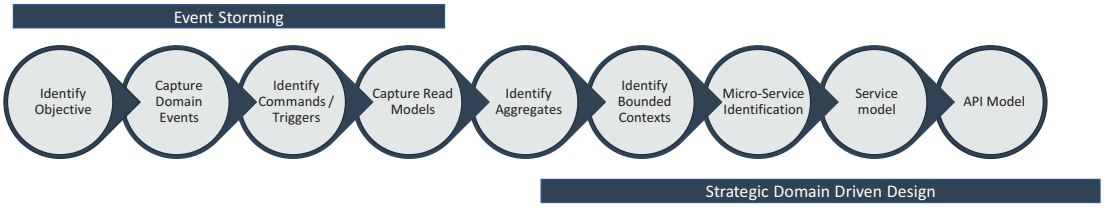


Figure 10-11. *Event storming steps*

The following sections cover the nine steps of the event storming process.

- *Identify objectives and capture domain events*: The event storming team can identify the domain scenarios and use cases and identify all the events that take place during the identified scenarios and then document and sequence events using sticky notes. The events must be in past tense like item purchased, invoice sent, invoice paid, etc.
- *Discussion*: Experts brainstorm by asking questions and clarifying details.
- *Identify commands and read models*: Identify the user and external systems that interact with the events.
- *Aggregates*: Identify aggregates by combining the events and commands
- *Bounded contexts*: Identify bounded contexts by using the events, commands, users, and systems identified in the event map.

Step 1: Identify the Objectives

In this step, as shown in Figure 10-12, you need to identify a domain scenario and use cases. In this example, I am choosing the auto insurance domain. Within the domain, there are value chains that are nothing more than the subdomains or departments in a portfolio of your organization. In the value chain, I am selecting the Policy Management value chain. Within the value chain, I am selecting the Quote & Policy Issuance use case to explain event storming further.

Within Quote & Policy Issuance, the objectives are quote generated, information provided, policy purchased, payment processed, account created, etc.

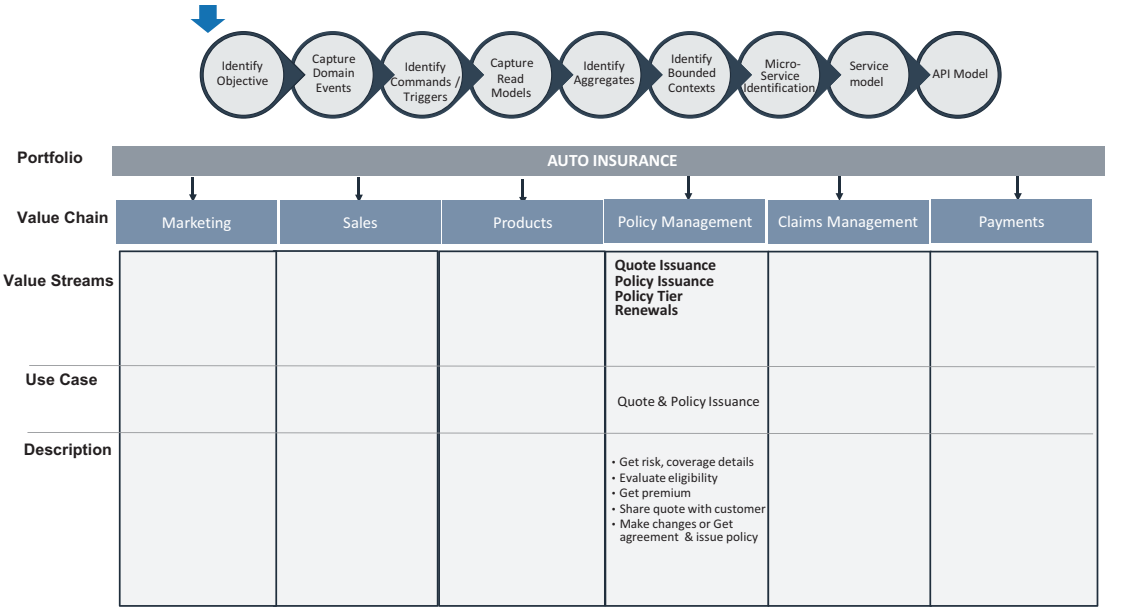


Figure 10-12. *Identify the objectives*

Step 2: Event Map: Capture Domain Events

In this step, as shown in Figure10-13, a team of experts discuss and identify domain events with a sequence. In the DT workshop, use orange sticky notes to identify all the events of an identified use case.

These are the key activities in this step:

- Document all events irrespective of minor or major occurrences for a given use case; events are written in the past tense.
- Rearrange events in a sequential order and resolve any. Group multiple events into larger, single events appropriately, and rearrange them based on time.
- Identify the actors responsible for each event.
- Capture questions, risk and warnings, assumptions, and conversation points.

Note Any activity in a use case is called an *event*.

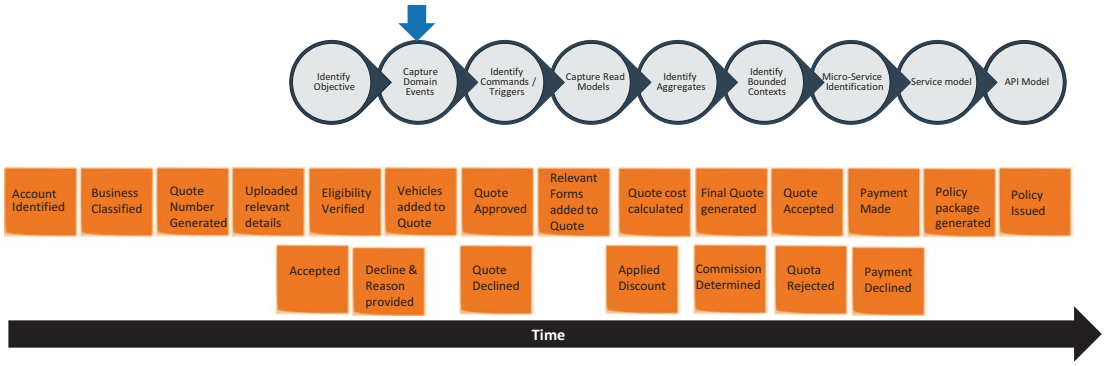


Figure 10-13. *Domain events*

Step 3: Event Map: Identify Commands, Triggers, and Read Models

With your events outlined, you can work on evaluating each event based on the behavior and what triggered this event. Without a trigger, there is no event. The trigger can be from external users or external systems or internal systems. The trigger of the event is noted as a command. Commands are documented by using blue sticky notes in the present tense and represent user interaction with the system.

Along with the commands, you need to add a user/role of the command and write it on a brown sticky note.

You need to capture the information about the commands such as the type of commands, how they are triggered etc., and write them on green sticky notes.

Figure 10-14 provides a clear view of the relationship between commands, events, users/roles, and read models.

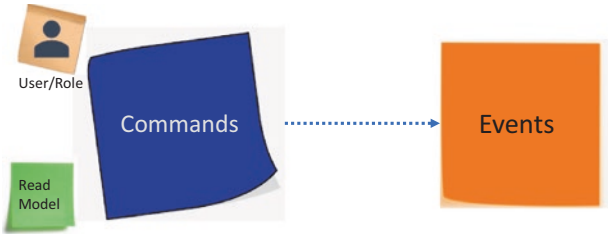


Figure 10-14. *Relationship between command and events*

Figure 10-15 shows the steps to group relevant events and identify the respective commands.

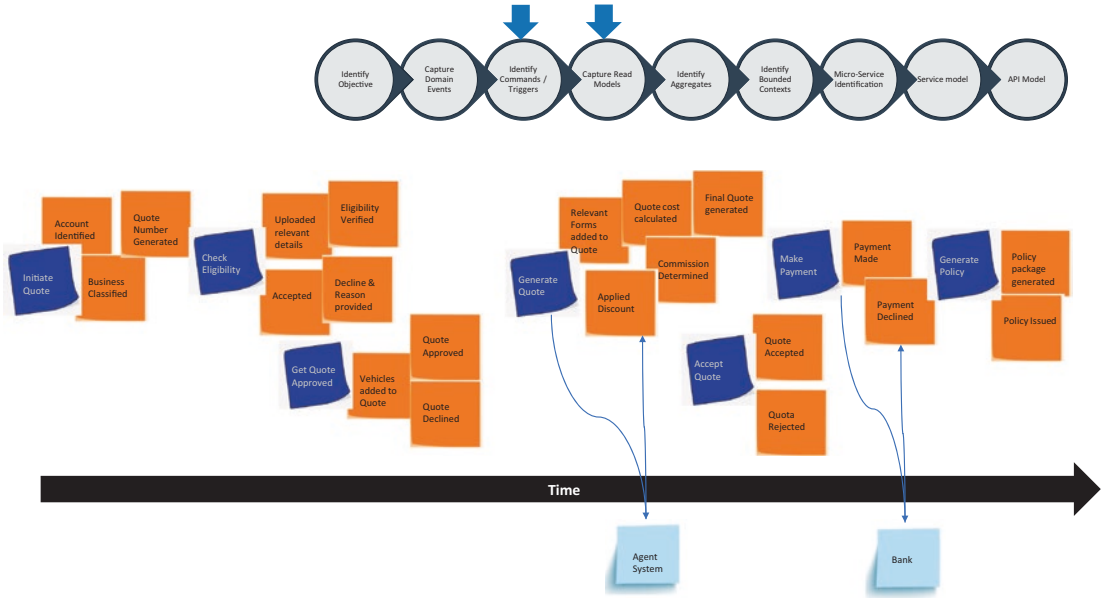


Figure 10-15. *Commands and events*

Step 4: Event Map: Identify Aggregators

An *aggregate* is a combination of domain events and commands that can be treated as a single unit. In an aggregate, one main domain event will be the aggregate root, and any reference from commands should go only to the aggregate root. The root can ensure the integrity of the aggregate as a whole. Don't mix a UML aggregate with a DDD aggregate. It is a domain concept, while collections are generic.

An aggregate consists of one or more entities and domain models that change together. You can consider them as a unit of data changes, and you need to consider the consistency of the entire aggregate for any changes. The aggregate helps you simplify the domain model by collecting multiple domain events under a single abstraction around domain variants and acts as the consistency and concurrency boundaries. The most important rule to define a boundary for your aggregate cluster is that the boundary should be based on domain invariants. Domain invariants are business rules that must always be consistent. The consistency boundary logically asserts that everything inside adheres to a specific set of business invariant rules no matter what operation is performed.

Entities inside the same aggregate should be highly cohesive, whereas entities outside aggregates are loosely coupled among other aggregates.

The aggregates have a local responsibility and receive commands and then emit domain events. The aggregates are documented by using yellow sticky notes in the form of a noun.

Figure 10-16 provides a clear view of the relationship between commands, events, users/roles, and read models and aggregates.

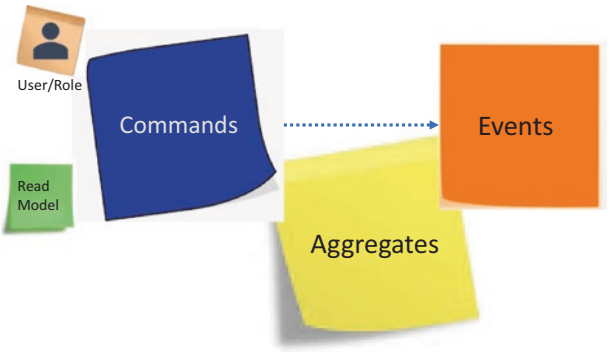


Figure 10-16. *Relationship between commands, events, and aggregates*

As shown in Figure 10-17, you can identify an aggregate from commands and events. The following are the rules to define an aggregate:

- Aggregates should be based on domain invariants.
- Aggregates should be modified with their invariants completely consistent with a single transaction.
- Aggregates represent domain concepts, not just a collection of domain events.
- Avoid having transaction across aggregates and consider them as a single unit of work.
- Try for smaller aggregates to support the "-ilities."

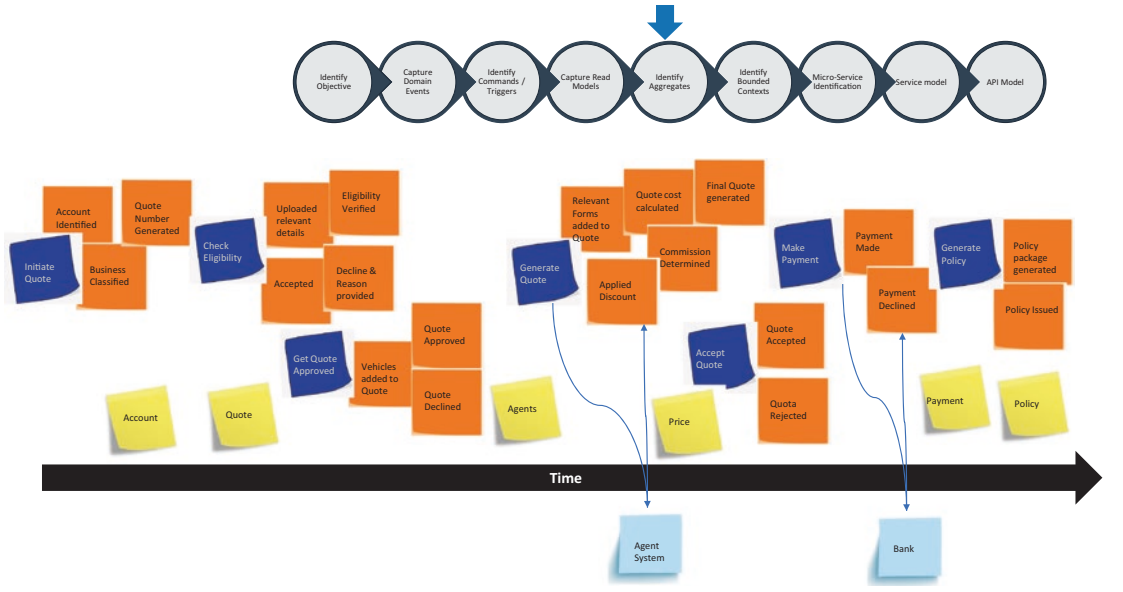


Figure 10-17. *Identification of aggregators*

Step 5: Context Map: Identify the Bounded Context

A *bounded context* is the logical boundary of a domain model that represents a particular subdomain of your system. It is the focus of the strategic design section to deal with domains and events. As I mentioned, the domain model represents the real things of the business, such as an account, insurance, policy, etc. It is the conceptual design of your system.

In an enterprise scenario, a bounded context is often based on ownership, with the bounded context being maintained by a team. For each bounded context, there will be a command and triggers along with the events produced. Typically, in strategic DDD, the bounded context is the last step you will define for a system. Each bounded context should be independent and owns its language and model. The rule of thumb is that each bounded context is a microservice.

Figure 10-18 shows the bounded context with domain events and commands.

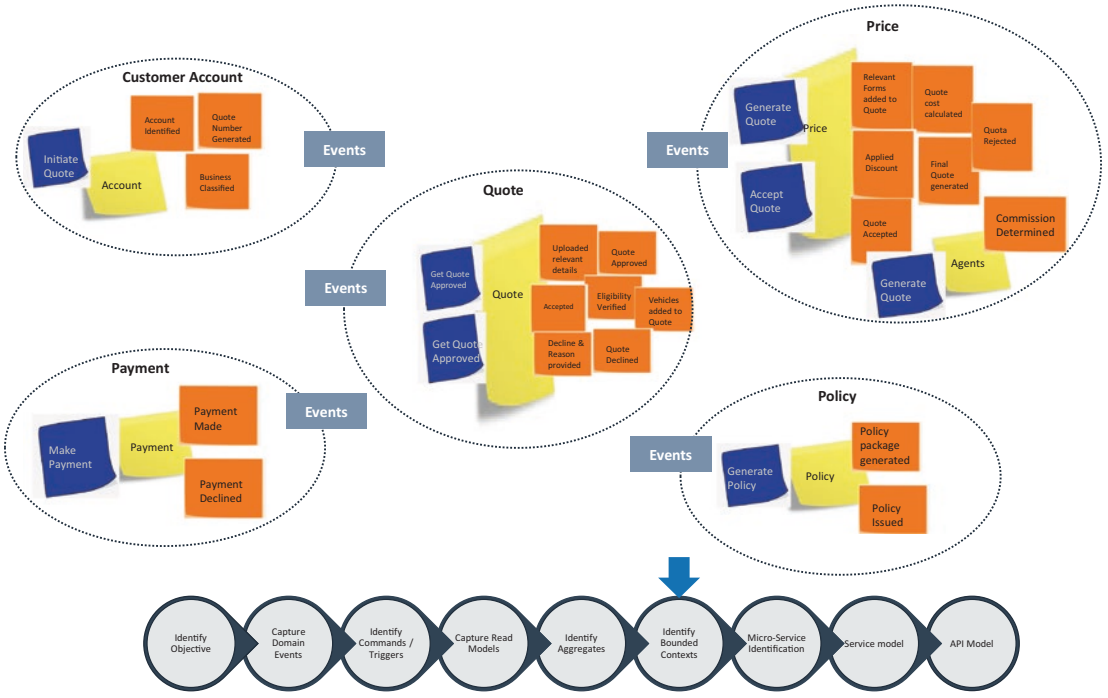


Figure 10-18. *Bounded context*

How do you identify a bounded context?

- Identify and collect the most meaningful domain events guided by domain knowledge based on business capabilities.
- Identify whether there's a clear cohesion required for certain domain events based on dependencies.
- A domain model has specific domain entities within a bounded context and delimits the applicability of the domain model and gives clear ownership to the pod team.
- Apply Conway's law to identify a bounded context; this law emphasizes that the system will reflect the social boundaries.
- Use the context mapping pattern to identify various contexts in your system and their boundaries.

How Does a Bounded Context Communicate?

As shown in Figure 10-19, a bounded context is loosely coupled with other bounded contexts; they interact only through synchronous or asynchronous communication by using REST and event protocols. You can refer to the communication details in Chapters 5 and 6.

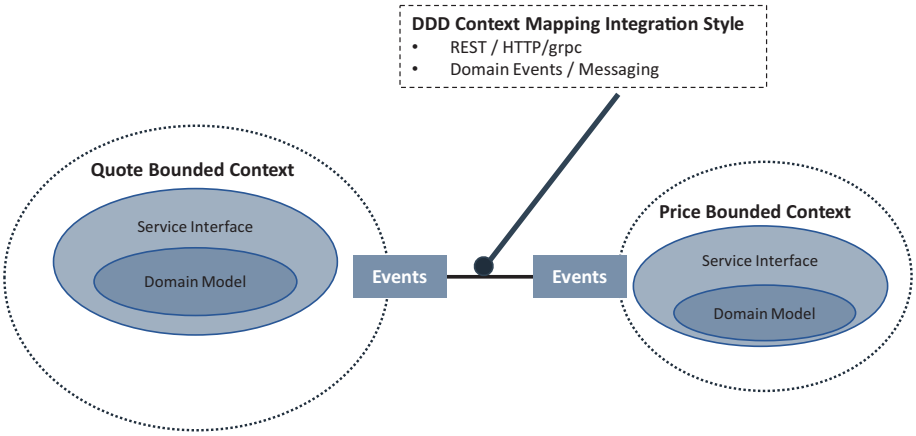


Figure 10-19. *Bounded context communication*

Ubiquitous Language

A ubiquitous language is a clearly defined language used for all discussion between domain experts, product teams, the architecture, engineers, etc. The ubiquitous language will also be used in the documentation, test cases, and code. Generally, each bounded context has its own ubiquitous language, and therefore a translation may be needed when communicating with another bounded context.

The following are the reasons why a ubiquitous language is needed:

- It is needed for understanding and communicating concepts in the domain in an unambiguous manner.
- It improves collaboration with domain experts to be more creative and valuable for all the teams.
- It is used for all the brainstorming between domain experts, product owners, architects, developers, testers, etc.
- It reveals the intention, not the implementation.

It helps to unite people in the project team and eliminate inaccuracies and contradictions. The domain model will evolve and will not end at a single meeting. You need to create a glossary of domain workshops to create a ubiquitous language.

Tactical Implementation of DDD

The goal of tactical DDD is to produce artifacts that are clearly defined and well understood by all team members. Identify the right thing to build.

- Tactical DDD occurs at a lower level typically within a team to support the service design.
- From a lean-agile perspective, this allows the team to share and align on what they need to align on.
- Create an integrated backlog and discuss epics, user stories, etc.
- Apply stories within a single bounded context.

Step 6: Microservices Identification

For microservices identification, look for entity and aggregators, as shown in Figure 10-20, which help you to identify the natural boundaries of the service. A general principle you need to consider is that a microservice should be no smaller than an aggregate and no larger than a bounded context.

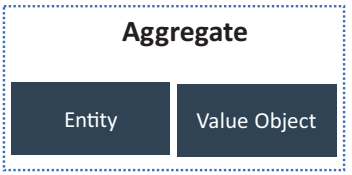


Figure 10-20. *Microservice identification*

Entity

An *entity* is an object with a unique identity that persists over time. For example, in an insurance quote, vehicle details and customer details would be entities.

- An entity has a unique identifier in the cloud native service, and the identifier is unique to the service and may span multiple bounded contexts.

- Objects have an identity that remains the same throughout the states of the software.
- An entity must be distinguished from other similar objects having the same attribute (e.g., customer account for an insurer).
- The attributes of an entity can change (mutable).

Value Objects

Value objects have no identity, and they are defined only by the values of the attributes. Value objects are the things within your model that convey meaning and functionality but have no uniqueness. These are used to pass parameters in messages between objects, and they are immutable. Attributes of value objects cannot change; they must be replaced with addresses, etc.

Aggregates

An aggregate defines a consistency boundary around one or more entities, and it is a cluster of entity and value objects. One entity is an aggregate of the root, and each aggregate is treated as one single unit that is retrieved and persisted together in a single transaction boundary. The root identity is global. The identities of entities inside are local, and the root is used for communication to the outside world. Internal objects cannot be changed outside the aggregate.

Domain Model to Microservices

In the previous section, I explained the bounded context, commands, and events, and explained how a bounded context is identified with a set of entities and aggregates.

As shown in Figure 10-21, here's an approach that you can use to derive microservices from the domain model:

- Let's start with a bounded context; in general, the functionality in a microservice should not span more than one bounded context. If you find a microservice that spans a bounded context, that's a sign that you may need to go back and refine your domain analysis.

- Look at the aggregates in your domain model; aggregates are often good candidates for microservices.
    - An aggregate must derive from commands and domain events.
    - An aggregate should have high function cohesion.
    - An aggregate is a boundary of persistence.
    - Aggregates should be loosely coupled.
- Finally, look at the "-ilities" and adopt Conway's law and an agile POD team structure for the ownership of a service. These factors may lead you to further decompose microservices.
- Each service must have a single responsibility and minimize transactions across services so there are no chatty calls between microservices.
- Each service is small enough that can build, manage, and destroy with small POD teams.
- Services have high cohesion inside and are loosely coupled outside.

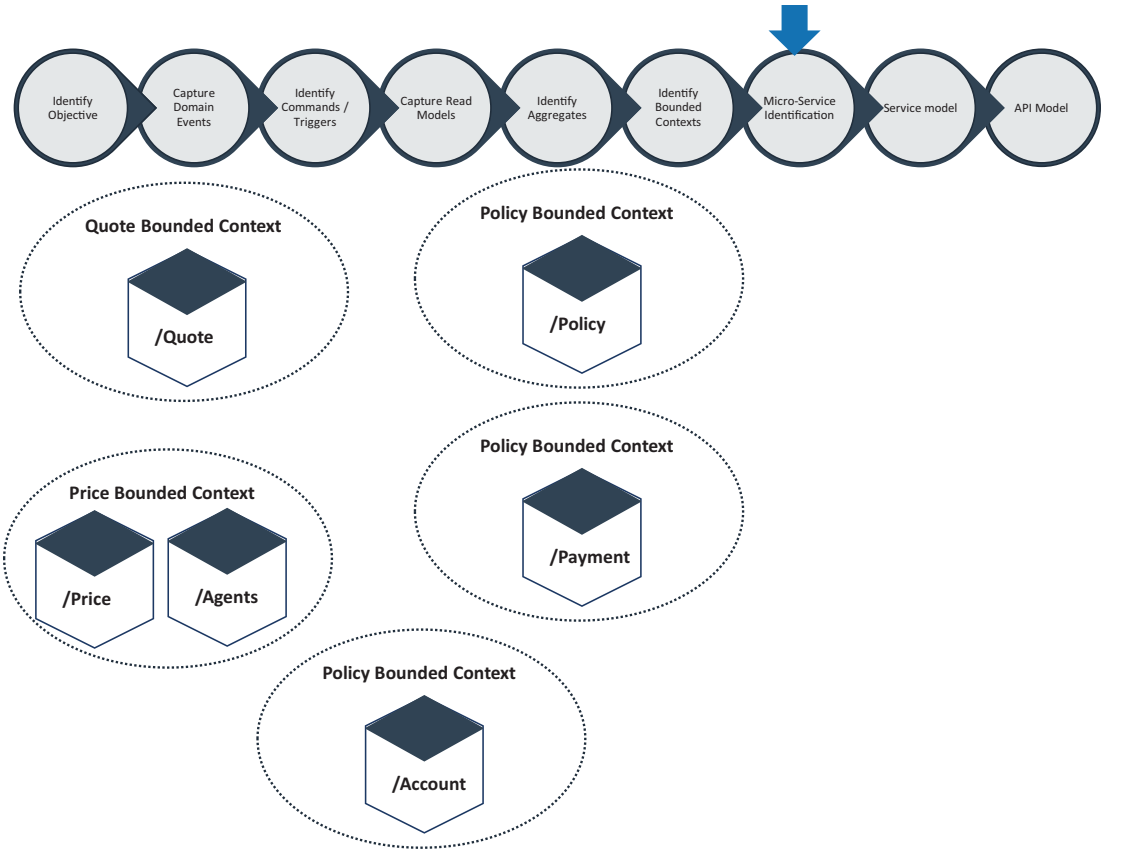


Figure 10-21. *Microservices model*

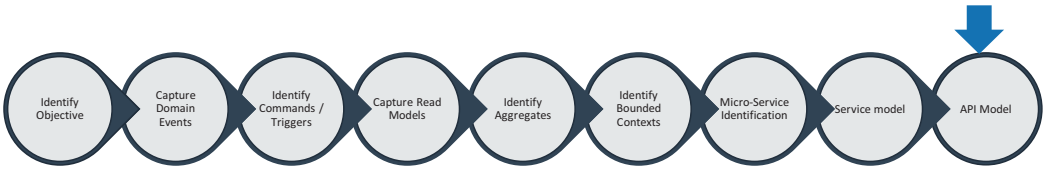
In the previous example, quote, account, payment, policy, agents and price are candidates of microservices.

API Model

A good API model has the same importance as a good design of microservices, because all data exchange between services occurs through APIs or events. APIs must be efficient to avoid creating chatty I/O. It is important to design and distinguish between public APIs and private APIs. Public APIs are exposed to the outside world, and private APIs are used for interservice communication or backend systems.

For public APIs, you need to consider REST over HTTP(s), and you need to consider various factors such as performance, backend systems protocols, etc. Depending on the granularity of services, interservice interaction can result in a lot of network traffic, and the service becomes I/O bound. For this reason, your services should be designed

with the appropriate granularity. Serialization speed and payload size become more important. You can consider REST over HTTP and gRPC, Apache Avro, and Apache Thrift. Figure 10-22 shows the sample APIs of the previous example.



Resource	Method	Operation	Parameter	Response	Description
/Account	GET	Get Account ID	Business Info	[Account Info]	Search accounts based on accountName
/Quote	POST	Create Quote Number	Basic Policy No	Quote Number	Stores Basic Policy info
/Quote {QNum}	GET	Referral determination		Referred/Declined /Cleared	Based on Quote number provided referral determination check will be made

Figure 10-22. *API model*

Value of Domain-Driven Design

There are multiple benefits of DDD.

- It is an extremely flexible approach to software.
- DDD takes on the domain model to decouple the business cases or legacy systems, and the technology will follow to realize the business model.
- DDD understands the customer values and perspective on the issues. The collaboration between domain experts, product teams, and technical teams can help to create a domain model with a ubiquitous language.
- The ubiquitous language used for each model provides clarity, precision, and commonality between all the stakeholders.

The Business Value of DDD

There are many reasons why a business finds value in DDD.

- The objective of DDD is to provide value to businesses by modeling the software from the business paradigm.
- DDD provides a clear understanding of how the business works and provides an understanding of how the business runs.
- The users of the systems are able to contribute starting from day one as a domain expert; this helps each service be built with a rich domain.
- This helps to focus on core domains.

Drawbacks of DDD

Because of the modular nature of DDD and the strict following of the domain, the software itself requires significant insulation, and isolation is one part of the development.

- There are unfamiliar processes and rules in the legacy system; they are difficult to identify and may miss some circumstances that become costly for the service design.
- It is effective as a domain, but you may not get an advantage by applying it for small, simple business domains.
- Ubiquitous is the common language in DDD. If one person doesn't get it, then it could represent a bad design.
- DDD has a learning curve in an enterprise.
- DDD adds time to the entire DDD process; sometimes the engineering lifecycle is very short, so the team ignores the DDD process to identify a service. That becomes costlier later for the service management.

Where DDD Is Not Useful

There are some common misconceptions of DDD.

- DDD is not a set of patterns that exists for repurposing, and it is not code-focused and not an object-oriented concept. If your project is more suited toward those things, we suggest using UML.
- DDD is not for every enterprise architecture design; we also suggest using either TOGAF or the Zachman framework.
- DDD is not a solution for everything. Different organizations will have difficulties that require a paradigm shift that simply cannot be solved by using DDD.
- DDD is not an architectural pattern or design pattern; it is about how to design your application with a focus on the domain.

Summary

DDD is a domain language that is designed to manage the creation and maintenance of a system, and it is a collection of patterns and principles that can be applied to service design to manage complexity. Its emphasis on the distillation of large problem domains into subdomains can reveal the core domain, which is the area of most value. Using a ubiquitous language across teams can better manage collaboration.

The best pace of technological change in decoupling is as follows:

- *Architectural design*: By adopting a cloud native architecture, you can build out systems with greater flexibility. You can shift to lean architecture, APIs, cloud-based service platforms, etc.
- *Engineering practice*: The value of architectural change accelerates when you embrace newer ways of working that speed up development and delivery like DevSecOps, automation, design thinking, etc.
- *Talent evolution*: You need to upskill resources at greater speed and scale than ever.

You must adopt the following actions for decoupling to a cloud native architecture:

- Decoupling data from legacy systems
- Decoupling applications from legacy infrastructure
- Decoupling tightly integrated systems into loosely coupled systems
- Decoupling organizations from traditional structures and measures
- Decoupling essential differentiation from unnecessary differentiation

Modernizing your enterprise is not straightforward. One way to gauge the need for modernization is to look at the current level of technical debt, essentially the money it would take to upgrade legacy systems.

To make modernization a reality, you must do the following:

- Adopt decoupling as a rational approach to focus on modernization in a way that gradually migrates systems away from legacy while effectively managing the costs and risk.
- Conduct an application assessment to identify recommendations that help you to draw a roadmap that offers transparency and reduces risk.
- Socialize a modernization approach like DDD and event storming that has a high success rate across industries.