Chapter 22. Introduction to the AKF Scale Cube

Ponder and deliberate before you make a move.  
—Sun Tzu

In Chapter 21, we referred several times to the AKF Scale Cube to highlight methods by which components of our architecture might be split into swim lanes or failure domains. In this chapter, we reintroduce the AKF Scale Cube. We developed the scale cube to help our clients think about how to split services, data, and transactions, and to a lesser degree, teams and processes.

The AKF Scale Cube

Imagine a cube drawn with the aid of a three-dimensional axis for a guide. We will call the point of intersection of our three axes the initial point, as referenced by the values \( x = 0, y = 0, \) and \( z = 0. \) Figure 22.1 shows this cube and these three axes. Each axis will describe a unique method of scaling products, processes, and teams.

![Figure 22.1 AKF Scale Cube with Axes](image)

The initial point, with coordinates of \((0, 0, 0)\), is the point of least scalability within any system. It consists of a single monolithic solution deployed on a single server. It might scale “up” with larger and faster hardware, but it won’t
As such, it will limit your growth to that which can be served by a single unit. In other words, your system will be bound by how fast the server runs the application in question and how well the application is tuned to use the server. We call work on any of the three axes a “split,” as in “an x-axis split” or a “y-axis split.”

Making modifications to your solution for the purposes of scaling moves you along one of the three axes. Equivalent effort applied to any axis will not always return equivalent results. For instance, a week of work applied to the x-axis might allow transaction growth to scale very well but not materially impact your ability to store, retrieve, and search through product-related information. To address such a storage or memory constraint, you might need to consider a week of work along the y-axis. Such y-axis splits might allow you to split information for the purposes of searching through it faster, but may hinder your efforts when the joining of information is required—for this, you might need to consider z-axis splits.

Choosing one axis does not preclude you from making use of other axes later. Recall from Chapter 19, Fast or Right?, that the cost to design splits is comparatively low relative to the cost of implementing and deploying those splits. As such, we want to think about how we would make use of each of the three axes in our cube in our designs. When we make choices about what to implement, we should select the splits that have the highest return (in terms of scale) for our effort and that meet our needs in terms of delivering scalability in time for customer demand.

In the following sections, we discuss the meaning of each of the axes at a very high level. In Chapter 23, Splitting Applications for Scale, and Chapter 24, Splitting Databases for Scale, we will dig deeper into the most common applications of each of these axes: splitting services and splitting databases.

The x-Axis of the Cube

The x-axis of the AKF Scale Cube represents cloning of services and data with absolutely no bias. Perhaps the easiest way to represent such a split is to think first in terms of people and organizations. Let’s first consider the days of yore in which typing pools handled the typing of meeting minutes, letters, internal memos, and so on. Use of the term pool dates back as far as 70 or more years as a means of identifying a single logical service (typing) distributed among several entities (in this case people). Work would be sent to the typing pool largely without a bias as to which individual typist performed the work. This distribution of work among clones is a perfect example of x-axis scalability.
Another people example to illustrate the x-axis might be found within the accounts receivable or accounts payable portion of your company’s finance organization. Initially, for small to medium companies that do not outsource this kind of work, the groups might consist of a few people, each of whom can perform all of the tasks within his or her area. The accounts payable staff can all receive bills and generate checks based on a set of processes and send those checks out or get them countersigned depending on the amount of the check. The accounts receivable staff is capable of generating invoices from data within the system, receiving checks, making appropriate journal entries, and depositing the checks. Each person can do all of the tasks, and it does not matter to whom the work goes.

Both of these examples illustrate the basic concept of the x-axis, which is the unbiased distribution of work across clones. Each clone can do the work of the other clones, and there is no bias with respect to where the work travels (other than individual efficiency). Each clone has the tools and resources to get the work done and will perform the work given to it as quickly as possible.

The x-axis seems great! When we need to perform more work, we just add more clones. Is the number of memoranda exceeding your current typing capacity? Simply add more typists! Is your business booming and there are too many invoices to create and too many payments coming in? Add more accounts receivable clerks! Why would we ever need any more axes?

Let’s return to our typing pool first to answer this question. Assume that to complete some of our memoranda, external letters, and notes, a typist needs to have certain knowledge. Suppose also that as the company grows, the services offered by the typing pool increase. The pool now performs some 100 different types and formats of services, and the work is not evenly distributed across these types of services. External client letters have several different formats that vary by the type of content included within the message, memoranda vary by content and intent, meeting notes vary by the type of meeting, and so on. Now an individual typist may get some work done very fast (the work that is most prevalent throughout the pool) but also be required to spend time looking up the less frequently encountered formatting, which in turn slows down the entire pipeline of work. As the type of work increases for any given service, more time may be spent trying to get work of varying sizes done, and the instruction set to accomplish this work may not be easily kept in any given typist’s head.

These are all examples of problems associated with the x-axis: It simply does not scale well with an increase in data, either as instruction sets or reference data. The same holds true if the work varies by the sender or receiver. For
instance, maybe vice presidents and senior executives get special formatting or are allowed to send different types of communication than directors of the company. Perhaps the sender uses special letterhead or stock. Maybe the receiver of the message triggers a variation in tone of communication or paper stock. Account delinquent letters, for instance, may require a special tone not referenced within the notes to be typed.

As another example, consider again our accounts receivable group. This group obviously performs a very wide range of tasks, from the invoicing of clients to the receipt of bills, the processing of delinquent accounts, and the deposit of funds into our bank accounts. The processes for each of these tasks grows as the company grows, and our controller will certainly want some specific process controls to exist so that money doesn’t errantly find its way out of the accounts receivable group and into an employee’s pocket before payday! This is another place where scaling for transaction growth alone is not likely to allow us to scale cost-effectively into a multibillion-dollar company. We will likely need to perform splits based on the services this group performs and/or the clients or types of clients they serve. These splits are addressed by the y- and z-axes of our cube, respectively.

The x-axis split tends to be easy to understand and implement and fairly inexpensive in terms of capital and time. Little additional process or training is necessary, and managers find it easy to distribute the work. Our people analogy holds true for systems as well, which we will see in Chapters 23 and 24. The x-axis works well when the distribution of a high volume of transactions or work is all that we need to do.

---

**Summarizing the x-Axis**

The x-axis of the AKF Scale Cube represents the cloning of services or data such that work can easily be distributed across instances with absolutely no bias.

The x-axis implementations tend to be easy to conceptualize and typically can be implemented at relatively low cost.

The x-axis implementations are limited by growth in the instructions to accomplish tasks and growth in the data necessary to accomplish tasks.

---

**The y-Axis of the Cube**

The y-axis of the AKF Scale Cube represents a separation of work responsibility
by either the type of data, the type of work performed for a transaction, or a combination of both. One way to view these splits is as a split by responsibility for an action. We often refer to these outcomes as service- or resource-oriented splits. In a y-axis split, the work for any specific action or set of actions, as well as the information and data necessary to perform that action, is split away from other types of actions. This type of split is the first split that addresses the monolithic nature of work and the separation of the same into either pipelined workflows or parallel processing flows. Whereas the x-axis is simply the distribution of work among several clones, the y-axis represents more of an “industrial revolution” for work: We move from a “job shop” mentality to a system of greater specialization, just as Henry Ford did with his automobile manufacturing assembly line. Rather than having 100 people creating 100 unique automobiles, with each person doing 100% of the tasks, we now have 100 unique individuals performing subtasks such as engine installation, painting, windshield installation, and so on.

Let’s return to our previous example of a typing service pool. In our x-axis example, we identified that the total output of our pool might be hampered as the number and diversity of tasks grew. Specialized information might be necessary based on the nature of typing work performed: An internal memorandum might take on a significantly different look than a memo meant for external readers, and meeting notes might vary by the type of meeting. The vast majority of the work might consist of letters to clients of a certain format and typed on a specific type of letterhead and bond. When a typist is presented with one of the 100 or so formats that represent only about 10% to 20% of the total work, he or she might have to stop and look up the appropriate format, grab the appropriate letterhead and/or bond, and so on. One approach to this situation might be to create much smaller pools specializing in some of the more common requests within this 10% to 20% of the total work and a third pool that handles the small minority of the remainder of the common requests. Both of these new service pools could be sized proportionally to the work.

The expected benefit of such an approach would be a significant increase in the throughput of the large pool representing a vast majority of the requests. This pool would no longer “stall” on a per-typist basis when presented with a unique request. Furthermore, for the next largest pool of typists, some specialization would happen for the next most common set of requests, and the output expectations would be the same; for those sets of requests, typists would be familiar with them and capable of handling them much more quickly than before. The remaining set of requests that represent a majority of formats but a
minority of request volume would be handled by the third pool; although throughput would suffer comparatively, it would be isolated to a smaller set of people who might also at least have some degree of specialization and knowledge. The overall benefit should be that throughput increases significantly. Notice that in creating these pools, we have also created a measure of fault isolation, as identified in Chapter 21, Creating Fault-Isolative Architectural Structures. Should one pool stall due to paper issues and such, the entire “typing factory” does not grind to a halt.

It is also easy to see how the separation of responsibilities would be performed within our running example of the accounts receivable department. Each unique action could become its own service. Invoicing might be split off into its own team or pool, as might payment receiving/journaling and deposits. We might further split late payments into its own special group that handles collections and bad debt. Each of these functions is associated with a unique set of tasks that require unique data, experience, and instructions or processes. By splitting them, we reduce the amount of information any specific person needs to perform his or her job, and the resulting specialization should allow us to perform processing faster. The y-axis industrial revolution has saved us!

Although the benefits of the y-axis are compelling, such splits tend to cost more than the simpler x-axis splits. The reason for the increase in cost is that to perform the y-axis split, very often some rework or redesign of process, rules, software, and the supporting data models or information delivery system is required. Most of us don’t think about splitting up the responsibilities of our teams or software when we are running a three-person company or operating a Web site running on a single server. Additionally, the splits themselves create some resource underutilization initially that manifests itself as an initial increase in operational cost.

The benefits of the y-axis are numerous, however. Although y-axis splits help manage the growth in transactions, they also help scale what something needs to know to perform those transactions. The data that is being operated upon as well as the instruction set to operate that data decreases, which means that people and systems can be more specialized, resulting in higher throughput on a per-person or per-system basis. Even a simple y-axis split will also give us a first level of fault isolation. In the accounts receivable example, if a member of a bad debts “pool” is sick for the day, only his or her service pool will be impacted.

**Summarizing the y-Axis**

The y-axis of the AKF Scale Cube represents separation of work by
The y-axis splits are easy to conceptualize but typically come at a slightly higher cost than the x-axis splits. The y-axis splits aid in scaling not only transactions, but also instruction size and data necessary to perform any given transaction.

The z-Axis of the Cube
The z-axis of the cube is a split biased most often by the requestor or customer or, alternatively, by something that we know about the requestor or customer. The bias here is focused on data and actions that are unique to the person for whom the request is being performed. While z-axis splits may or may not address the monolithic nature of instructions, processes, or code, they very often do address the monolithic nature of the data necessary to perform these instructions, processes, or code.

To perform a z-axis split of our typing service pool, we might look at both the people who request work and the people to whom the work is being distributed. In analyzing the request work, we can look at segments or classes of groups that might require unique work or represent exceptional work volume. It’s likely the case that executives represent a small portion of our total employee base, yet account for a majority or supermajority of the work for internal distribution. Furthermore, the work for these types of individuals might be somewhat unique in that executives are allowed to request more types of work to be performed. Maybe we will limit internal memoranda to executive requests, or say that personal customer notes might be requested only by an executive. A specialist pool of typists might best serve this unique volume and type of work. We might also dedicate one or more typists to the CEO of the company, who likely has the greatest number and variety of requests. All of these are examples of z-axis splits.

In our accounts receivable department, we might decide that some customers require specialized billing, payment terms, and interaction unique to the volume of business they do with us. We might dedicate a group of our best financial account representatives and even a special manager to one or more of these customers to handle their unique demands. In so doing, we would reduce the amount of knowledge necessary to perform the vast majority of our billing functions for the majority of our customers while creating account specialists for our most valuable customers. We would expect these actions to increase the throughput of our standard accounts group, as they need not worry about special
terms. The relative throughput for special accounts should also increase, as these individuals specialize in that area and are familiar with the special processes and payment terms.

Although z-axis splits are sometimes the most costly for companies to implement, the returns from them (especially from a scalability perspective) are usually phenomenal. Specialized training in the previous examples represents a new cost to the company, and this training is an analog to the specialized set of services one might need to create within a systems platform. Data separation can become costly for some companies, but when performed can be amortized over the life of the platform or the system.

An additional benefit that z-axis splits create is the ability to separate services by geography. Want to locate your accounts receivable group closer to the accounts they support to decrease mail delays? Easy to do! Want your typing pool to be close to the executives and people they support to limit interoffice mail delivery (remember these are the days before email)? Also simple to do!

---

**Summarizing the z-Axis**

The z-axis of the AKF Scale Cube represents separation of work by customer or requestor.

Like x- and y-axis splits, z-axis splits are easy to conceptualize, but very often can be difficult and costly to implement for companies.

The z-axis splits aid in scaling transactions and data and may aid in scaling instruction sets and processes if implemented properly.

---

**Putting It All Together**

Why would we ever need more than one, or maybe two, axes of scale within our platform or organizations? The answer is that your needs will vary by your company’s current size and expected annual growth. If you expect your organization to stay small and grow slowly, you may never need more than one axis of scale. In contrast, if you expect to grow quickly, or if growth is unexpected and violent, you are better off having planned for that growth in advance. Figure 22.2 depicts our cube, the axes of the cube, and the appropriate labels for each of the axes.
The $x$-axis of scale is very useful and easy to implement, especially if you have stayed away from creating state within your system or team. You simply clone the activity among several participants. Scaling along the $x$-axis starts to fail, however, when you have a lot of different tasks requiring significantly different information from many potential sources. Fast transactions start to run at the speed of slow transactions, and everything starts to work suboptimally.

---

**State Within Applications and the $x$-Axis**

You may recall from Chapter 12 that we briefly defined stateful systems as “those in which operations are performed within the context of previous and subsequent operations.” We indicated that state very often drives up the cost of the operations of systems, as most often the state (previous and subsequent calls) is maintained within the application or a database associated with the application. The associated data often increases memory utilization, storage utilization, and potentially database usage and licenses.

Stateless systems may allow us to break the affinity between a single user and a single server. Because subsequent requests can go to any server clone, the $x$-axis becomes even easier to implement. The lack of affinity between customer and server means that we need not design systems specific to any type of customer and so forth. Systems are now free to be
more uniform in composition. This topic will be covered in more detail in Chapter 26, Asynchronous Design for Scale.

The y-axis helps to solve that problem by isolating transaction type and speed to systems and people specializing in that area of data or service. Slower transactions are now bunched together, but because the data set has been reduced relative to the x-axis only example, they run faster than they had previously. Fast transactions also speed up because they are no longer competing for resources with the slower transactions and their data set has been reduced. Monolithic systems are reduced to components that operate more efficiently and can scale for data and transaction needs.

The z-axis not only helps us scale transactions and data, but may also help with monolithic system deconstruction. Furthermore, we can now move teams and systems around geographically and start to gain benefits from this geographic dispersion, such as disaster recovery.

Looking at our pool of typists, we can separate the types of work that the typists perform based on the actions involved. We might create a customer-focused team responsible for general customer communication letters, an internal memos team, and a team focused on meeting minutes—all of these are examples of the y-axis. Each team is likely to have some duplication to allow for growth in transactions within that team, which is an example of x-axis scale. Finally, we might decide that some members of the team should specialize to handle specific customers or requestors such as an executive group. Although this is a z-axis split, these teams may also have specialization by task (y-axis) and duplication of team members (x-axis). Aha! We’ve put all three axes together.

Think of how you not only addressed a problem of scale with this example, but also reduced some of the risk in introducing change. If you have a new process you want to try, you can introduce it to a group within a service pool (along the x-axis), to an entire functional team (along the y-axis), or to a region (along the z-axis). You can see how the new process works with a limited group of people, and decide after measuring it whether to roll it out to the other teams, to tweak it before introducing it everywhere, or to roll it back if it isn’t working as expected.

For our accounts receivable department, we have split the groups based on the invoicing, receiving, and deposits activities, all of which are y-axis splits. Each group has multiple members performing the same task, which is an x-axis split.
We have created special separation of these teams focused on major accounts and recurring delinquent accounts, and each of these specialized teams (a z-axis split) has further splits based on function (y-axis) and duplication of individuals (x-axis).

**AKF Scale Cube Summary**

Here is a summary of the three axes of scale:

- The *x*-axis represents the distribution of the same work or mirroring of data across multiple entities.
- The *y*-axis represents the distribution and separation of work responsibilities or data meaning among multiple entities.
- The *z*-axis represents the distribution and segmentation of work by customer, customer need, location, or value.

Hence, *x*-axis splits are mirror images of functions or data, *y*-axis splits separate data based on data type or type of work, and *z*-axis splits separate work by customer, location, or some value-specific identifier (e.g., a hash or modulus).

**When and Where to Use the Cube**

We will discuss the topic of where and when to use the AKF Scale Cube in Chapters 23, Splitting Applications for Scale, and 24, Splitting Databases for Scale. That said, the cube is a tool and reference point for nearly any discussion around scalability. You might make a representation of it within your scalability, 10×, or headroom meetings—a process that was discussed in Chapter 11, Determining Headroom for Applications. The AKF Scale Cube should also be presented during Architecture Review Board (ARB) meetings, as discussed in Chapter 13, if you adopt a principle requiring the design of more than one axis of scale for any major architectural effort. It can serve as a basis for nearly any conversation around scale, because it helps to create a common language among the engineers of an organization. Rather than talking about specific approaches, teams can focus on concepts that might evolve into any number of approaches.

You might consider requiring footnotes or light documentation indicating the type of scale for any major design within the joint architecture design (JAD) process introduced in Chapter 13, Joint Architecture Design and Architecture Review Board. The AKF Scale Cube can also come into play during problem resolution and postmortems in identifying how intended approaches to scale did
or did not work as expected and suggesting how to fix them in future endeavors.

Agile teams can use the AKF Scale Cube to use a single “scale” language within the team. Use it during Agile designs and have someone ask the question, “How does this solution scale, and along which axis does it scale?”

The AKF Scale Cube is a tool best worn on your tool belt rather than placed in your toolbox. It should be carried at all times, as it is lightweight and can add significant value to you and your team. If referenced repeatedly, it can help to change your culture from one that focuses on specific fixes to one that discusses approaches and concepts to help identify the best potential fix. It can switch an organization from thinking like technicians to acting like engineers.

**Conclusion**

This chapter reintroduced the concept of the AKF Scale Cube. This cube has three axes, each of which focuses on a different approach toward scalability. Organizational construction may be used as an analogy for systems to help better reinforce the approach of each of the three axes of scale. The cube is constructed such that the initial point \((x = 0, y = 0, z = 0)\) is a monolithic system or organization (single person) performing all tasks with no bias based on the task, customer, or requestor.

Growth in people or systems performing the same tasks represents an increase in the \(x\)-axis. This axis of scale is easy to implement and typically comes at the lowest cost, but it suffers when the number of types of tasks or data necessary to perform those tasks increases.

A separation of responsibilities based on data or the activity being performed is growth along the \(y\)-axis of our cube. This approach tends to come at a slightly higher cost than \(x\)-axis growth, but also benefits from a reduction in the data necessary to perform a task. Other benefits of such an approach include some fault isolation and an increase in throughput for each of the new pools based on the reduction of data or instruction set.

A separation of responsibility based on customer or requestor represents growth along the \(z\)-axis of scale. Such separation may allow for reduction in the instruction set for some pools and almost always reduces the amount of data necessary to perform a task. The result is that both throughput and fault isolation are often increased. The cost of \(z\)-axis splits tends to be the highest of the three approaches in most organizations, although the return is also huge. The \(z\)-axis split also allows for geographic dispersion of responsibility.

Not all companies need all three axes of scale to survive. Some companies
may do just fine with implementing the x-axis. Extremely high-growth companies should plan for at least two axes of scale and potentially all three. Remember that planning (or designing) and implementing are two separate functions.

Ideally the AKF Scale Cube, or a construct of your own design, will become part of your daily tool set. Using such a model helps reduce conflict by focusing on concepts and approaches rather than specific implementations. If added to JAD, ARB, and headroom meetings, it helps focus the conversation and discussion on the important aspects and approaches to growing your technology platform.

**Key Points**

- The AKF Scale Cube offers a structured approach and concept for discussing and solving scale. The results are often superior to a set of rules or implementation-based tools.
- The x-axis of the AKF Scale Cube represents the cloning of entities or data and an equal unbiased distribution of work across them.
- The x-axis tends to be the least costly to implement, but suffers from constraints in instruction size and data set.
- The y-axis of the AKF Scale Cube represents separation of work biased by activity or data.
- The y-axis tends to be more costly than the x-axis but solves issues related to instruction size and data set size in addition to creating some fault isolation.
- The z-axis of the AKF Scale Cube represents separation of work biased by the requestor or person for whom the work is being performed.
- The z-axis of the AKF Scale Cube tends to be the most costly to implement but very often offers the greatest scale. It resolves issues associated with data set size and may or may not solve instruction set issues. It also allows for global distribution of services.
- The AKF Scale Cube can be an everyday tool used to focus scalability-related discussions and processes on concepts. These discussions result in approaches and implementations.
- Designing and implementing are two different functions. The cost to design is relatively much lower than the cost to implement. Therefore, if you design to scale along the three dimensions of the AKF Scale Cube
early in the product development life cycle, you can choose to implement or deploy later when it makes sense for your business.

- ARB, JAD, Agile design, and headroom are all process examples where the AKF Scale Cube might be useful.
Chapter 23. Splitting Applications for Scale

Whether to concentrate or to divide your troops must be decided by circumstances.
—Sun Tzu

The previous chapter introduced the model by which we describe splits to allow for nearly infinite scale. Now we’ll apply those concepts to our real-world product needs. To do this, we’ll separate the product into pieces that address our application and service offerings (covered in this chapter) and the splits necessary to allow our storage and databases to scale (covered in the next chapter). The same model and set of principles hold true for both approaches, but the implementation varies enough that it makes sense for us to address them in two separate chapters.

The AKF Scale Cube for Applications

Whether applied to databases, applications, storage, or even organizations, the underlying meaning of the AKF Scale Cube does not change. However, given that we will now use this tool to accomplish a specific purpose, we will add more specificity to the axes. These additional descriptions remain true to the original but provide greater clarity for the architecting of applications to allow for greater scalability. Let’s first start with the AKF Scale Cube from the end of Chapter 22.

In Chapter 22, we defined the x-axis of our cube as the cloning of services and data with absolutely no bias. In the x-axis approach to scale, the only thing that is different between one system and 100 systems is that the transactions are evenly split between those 100 systems as if each was a single instance capable of handling 100% of the original requests rather than the 1% that they actually do handle. We will rename our x-axis as horizontal duplication/cloning of services to make it more obvious how we will apply this to our architecture efforts.

The y-axis is represented as a separation of work responsibility by either the type of data, the type of work performed for a transaction, or a combination of both. We most often describe this as a service-oriented split within an application; as such, we will now label this axis as a split by function or service. Here, “function” and “service” are indicative of the actions performed by your platform, but they can just as easily be resource-oriented splits, such as those
based on the object upon which an action is being taken. A function- or service-oriented split should be thought of as occurring along action or “verb” boundaries, whereas a resource-oriented split most often takes place along “noun” boundaries. We’ll describe these splits later in this chapter.

The z-axis focuses on data and actions that are unique to the person or system for which the request is being performed. We sometimes refer to the z-axis as being a “lookup-oriented” split in applications. The “lookup” term indicates that users or data are subject to a non-action-oriented bias that is represented somewhere else within the system. We store the relationships of users to their appropriate split or service somewhere, or determine an algorithm such as a hash or modulus of a user ID that will reliably and consistently send us to the right location set of systems to get the answers for the set of users in question. Alternatively, we may apply an indiscriminate function to the transaction (e.g., a modulus or hash) to determine where to send the transaction.

The new AKF Scale Cube for applications now looks like Figure 23.1.

*Figure 23.1 AKF Application Scale Cube*

The x-Axis of the AKF Application Scale Cube
The x-axis of the AKF Application Scale Cube represents cloning of services with absolutely no bias. As described previously, if we have a service or platform that is scaled using the x-axis alone and consisting of N systems, each of the N systems can respond to any request and will give exactly the same answer as the other (N – 1) systems. There is no bias based on service performed, customer, or any other data element. For example, the login functionality exists in the same location and application as the shopping cart, checkout, catalog, and search functionality. Regardless of the request, it is sent to one of the N systems that constitute our x-axis split.

The x-axis approach is simple to implement in most cases. You simply take exactly the same code that existed in a single instance implementation and put it on multiple servers. If your application is not “stateful,” simply load balance all of the inbound requests to any of the N systems. If you are maintaining data associated with user state or otherwise require persistence from a user to an application or Web server (i.e., the application is “stateful”), the implementation is slightly more difficult. In the cases where persistency or state (or persistency resulting from the need for state) is necessary, a series of transactions from a single user are simply pegged to one of the N instances of the x-axis split. This can be accomplished with session cookies from a load balancer. Additionally, as we will discuss in more detail in Chapter 26, Asynchronous Design for Scale, certain methods of centralizing session management can be used to allow any of N systems to respond to an individual user’s request without requiring persistency to that system.

The x-axis split has several benefits and drawbacks. Most notably, this split is relatively simple to envision and implement. The x-axis also allows for near-infinite scale from a number of transactions perspectives. When your applications or services are hosted, it does not increase the complexity of your hosting environment. Drawbacks of the x-axis approach include the inability of this split to address scalability from a data/cache perspective or instruction complexity perspective.

As stated, x-axis splits are easy to envision and implement. As such, when you face the prospect of developing a quick solution to any scale initiative, x-axis splits should be one of the first options that you consider. Because it is generally easy to clone services, the cost impact in terms of design expense and implementation expense is low. Furthermore, the additional time-to-market cost to release functionality with an x-axis split is generally low compared to other implementations, as you are simply cloning the services in question and can easily automate this task.
In addition, x-axis splits allow us to easily scale our platforms with the number of inbound transactions or requests. If you have a single user or small number of users who grow from making 10 requests per second to 1,000 requests per second, you need add only roughly 100 times the number of systems or cloned services to handle the increase in requests.

Finally, the team responsible for managing the services of your platform does not need to worry about a vast number of uniquely configured systems or servers. Every system with an x-axis split is roughly equivalent to every other system with the same split. Configuration management of all servers is relatively easy to perform, and new service implementation is as simply as cloning an existing system or generating a machine instance or virtual machine. Configuration files likely do not vary, and the only things the Agile team needs to be concerned about are the total number of systems in an x-axis implementation and whether each is getting an appropriate amount of traffic. In IaaS cloud environments, this approach is used for auto-scaling.

Although x-axis splits scale well with increased transaction volumes, they do not address the problems incurred with increasing amounts of data. Consider the case where a product must cache a great deal of data to serve client requests. As that data volume grows, the time to serve any given request will likely increase, which is obviously bad for the customer experience. Additionally, the product may become constrained on the server or application itself if the data size becomes too unwieldy. Even if caching isn’t required, the need to search through data on other storage or database systems will likely expand as the customer base and/or product catalog increases in size.

In addition, x-axis splits don’t address the complexity of the software implementing your system, platform, or product. Everything in an x-axis split alone is assumed to be monolithic in nature; as a result, applications will likely start to slow down as servers page instruction/execution pages in and out of memory to perform different functions. As a product becomes more feature rich, monolithic applications slow down and become more costly and less easily scaled, either as a result of this instruction complexity or because of the data complexity mentioned earlier. Engineering teams start to lose velocity or throughput as the monolithic code base begins to become more complicated to understand.

**Summarizing the Application x-Axis**

The x-axis of the AKF Application Scale Cube represents the cloning of an application or service such that work can easily be distributed across
instances with absolutely no bias.

In general, x-axis implementations are easy to conceptualize and typically can be implemented at relatively low cost. They are the most cost-effective way of scaling transaction growth. They can be easily cloned within your production environment from existing systems or “jumpstarted” from “golden master” copies of systems. They do not tend to increase the complexity of your operations or production environment.

On the downside, x-axis implementations are limited by the growth of a monolithic application, which tends to slow down the processing of transactions. They do not scale well with increases in data or application size. They do not allow engineering teams to scale well, as the code base is monolithic and can become comparatively complex.

The y-Axis of the AKF Application Scale Cube

The y-axis of the scale cube represents a separation of work responsibility within your application. We most frequently think of it in terms of functions, methods, or services within an application. The y-axis split addresses the monolithic nature of an application by separating that application into parallel or pipelined processing flows. A pure x-axis split would have N instances of the exact same application performing exactly the same work on each instance. Each of the N instances would receive 1/Nth of the work. In a y-axis split, we might take a single monolithic application and split it up into Y distinct services, such as login, logout, read profile, update profile, search profiles, browse profiles, checkout, display similar items, and so on.

Not surprisingly, then, y-axis splits are more complicated to implement than x-axis splits. At a very high level, it is often possible to implement a y-axis split in production without actually splitting the code base itself, although the benefits derived from this approach are limited. You can do this by cloning a monolithic application and deploying it on multiple physical or virtual servers.

As an example, let’s assume that you want to have four unique y-axis split servers, each serving one-fourth of the total number of functions within your site. One server might serve login and logout functionality, another read and update profile functionality, another “contact individual” and “receive contacts,” and yet another all of the other functions of your platform. You may assign a unique URL or URI to each of these servers, such as login.akfpartners.com and contacts.akfpartners.com, and ensure that any of the functions within the appropriate grouping always get directed to the server (or pool of servers) in
question. This is a good first approach to performing a split and helps work out the operational kinks associated with splitting applications. Unfortunately, it doesn’t give you all of the benefits of a full y-axis split made within the code base itself.

Most commonly, y-axis splits are implemented to address the issues associated with a code base and data set that have grown significantly in complexity or size. They also help scale transaction volume, as in performing the splits you must add virtual or physical servers. To get the most benefit from a y-axis split, the code base itself needs to be split up from a monolithic structure to a series of individual services that constitute the entire platform.

Operationally, y-axis splits help reduce the time necessary to process any given transaction as the data and instruction sets that are being executed or searched are smaller. Architecturally, y-axis splits allow you to grow beyond the limitations that systems place on the absolute size of software or data. In addition, y-axis splits aid in fault isolation as identified within Chapter 21, Creating Fault-Isolative Architectural Structures; a failure of a given service will not bring down all of the functionality of your platform.

From an engineering perspective, y-axis splits allow you to grow your organization more easily by focusing teams on specific services or functions within your product. For example, one team might be dedicated to the search and browse functionality, another team to the development of an advertising platform, yet another team to account functionality, and so on. New engineers get up to speed faster because they are dedicated to a specific section of functionality within your system. More experienced engineers will become experts at a given system and as a result can produce functionality within that system faster. The data elements upon which any y-axis split works will likely be a subset of the total data on the site; as such, engineers will better understand the data with which they are working and be more likely to make better choices in creating data models.

Of course, y-axis splits also have drawbacks. They tend to be more costly to implement in terms of engineering time than x-axis splits because engineers need to rewrite—or at the very least disaggregate—services from the monolithic application. In addition, the operations and infrastructure teams now need to support more than one configuration of server. This, in turn, might mean that the operations environment includes more than one class or size of server so as to utilize the most cost-efficient system for each type of transaction. When caching is involved, data might be cached differently in different systems, although we highly recommend that a standard approach to caching be shared across all of
the splits. URL/URI structures will grow, and when referencing other services, engineers will need to understand the current structure and layout of the site or platform to address each of the services.

**Summarizing the Application y-Axis**

The y-axis of the AKF Application Scale Cube represents separation of work by service or function within the application. Splits of the y-axis are meant to address the issues associated with growth and complexity in the code base and data sets. The intent is to create fault isolation as well as to reduce response times for y-axis split transactions.

The y-axis splits can scale transactions, data sizes, and code base sizes. They are most effective in scaling the size and complexity of your code base. They tend to cost a bit more than x-axis splits because the engineering team needs to rewrite services or, at the very least, disaggregate them from the original monolithic application.

**The z-Axis of the AKF Application Scale Cube**

The z-axis of the Application Scale Cube is a split based on a value that is “looked up” or determined at the time of the transaction; most often, this split is based on the requestor or customer of the transaction. The requestor and the customer may be completely different people. The requestor, as the name implies, is the person submitting a request to the product or platform, whereas the customer is the person who will receive the response or benefit of the request. Note that these are the most common implementations of the z-axis, but not the only possible implementation. For the z-axis split to be valuable, it must help partition not only transactions, but also the data necessary to operate on those transactions. A y-axis split helps us scale by reducing instructions and data necessary to perform a service; a z-axis split attempts to do the same thing through non-service-oriented segmentation.

To perform a z-axis split, we look for similarities among groups of transactions across several services. If a z-axis split is performed in isolation from the x- and y-axes, each split will be a single monolithic instance of a product. The most common implementation of a z-axis split involves segmenting the identified solution N ways, where each of the N implementations is the same code base. In some cases, however, deployments may contain a superset of capabilities. As an example, consider the case of the “freemium” business model, where a subset of services is free and perhaps supported by advertising,
while a larger set of services requires a license fee for usage. Paying customers
may be sent to a separate server or set of servers with a broader set of
capabilities.

How do we get benefits with a z-axis split if we have the same monolithic
code base across all instances? The answer lies in the activities of the individuals
interacting with those servers and the data necessary to complete those
transactions. Many applications and sites today rely on such extensive caching
that it becomes nearly impossible to cache all the necessary data for all potential
transactions. Just as the y-axis split helped us cache some of this data for unique
services, so does the z-axis split help us cache data for specific groups or classes
of transactions biased by user characteristics.

The benefits of a z-axis split are an increase in fault isolation, transactional
scalability, and cache-ability of objects necessary to complete our transactions.
You might offer different levels of service to different customers, though to do so
you might need to layer a y-axis split within a z-axis split. The end results we
would expect from these splits are higher availability, greater scalability, and
faster transaction processing times.

The z-axis, however, does not help us as much with code complexity, nor does
it improve time to market. Furthermore, we add some operational complexity to
our production environment; we now need to monitor several different systems
with similar code bases performing similar functions for different clients.
Configuration files may differ as a result, and systems may not be easily moved
once configured depending on your implementation.

Because we are leveraging characteristics unique to a group of transactions,
we can also improve our disaster recovery plans by geographically dispersing
our services. We can, for instance, locate services closer to the clients using or
requesting those services. With a sales lead system, we could put several small
companies in one geographic area on a server close to those companies; for a
large company with several sales offices, we might split that company into
several sales office systems spread across the company and placed near the
offices in question.

Another example of a z-axis split would be separating products by SKU (stock
keeping unit) or product number. These are typically numeric IDs such as
“0194532” or alphanumeric labels such as “SVN-JDF-045.” For search
transactions on a typical ecommerce site, for instance, we could divide our
search transactions into three groups, each serviced by a different search engine.
The three groups might consist of SKUs starting with numbers 0–3, 4–6, and 7–
9, respectively. Alternatively, we could separate our searches by product category—for example, cookware in one group, books in another group, and jewelry in a third search group.

The z-axis also helps to reduce risk. Whether within a continuous or phased delivery model, deployment of new solutions to a segment of users limits the impacts of new changes on the entire population of users.

---

**Summarizing the Application z-Axis**

The z-axis of the AKF Application Scale Cube represents separation of work based on attributes that are looked up or determined at the time of the transaction. Most often, these are implemented as splits by requestor, customer, or client.

Of the three types of splits, z-axis splits tend to be the most costly to implement. Although software does not necessarily need to be disaggregated into services, it does need to be written such that unique pods can be implemented. Very often, a lookup service or deterministic algorithm will need to be written for these types of splits.

The z-axis splits aid in scaling transaction growth, scaling instruction sets, and decreasing processing time (the last by limiting the data necessary to perform any transaction). The z-axis is most effective at scaling growth in customers or clients. It can aid with disaster recovery efforts, and limit the impact of incidents to only a specific segment of customers.

---

**Putting It All Together**

The observant reader has probably figured out that we are about to explain why you need multiple axes of scale and not just single-axis splits. We will work backward through the axes and explain the problems with implementing them in isolation.

A z-axis only implementation has several problems when applied in isolation. To better understand these problems, let’s assume the previous case where you make $N$ splits of your customer base in a sales lead tracking system. Because we are implementing only the z-axis here, each instance is a single virtual or physical server. If it fails for hardware or software reasons, the services for that customer or set of customers become completely unavailable. That availability problem alone is reason enough for us to implement an x-axis split for each of
our z-axis splits. If we split our customer base $N$ ways along the z-axis, with each of the $N$ splits having at least $1/N$th of our customers initially, we would put at least two “cloned” or x-axis servers in each of the $N$ splits. This ensures that if a server fails, we can still service the customers in that pod. Reference Figure 23.2 as we discuss this implementation further.

![Figure 23.2 Example: z- and x-Axes Split](image)

It is likely more costly for us to perform continued customer-oriented splits to scale our transactions than it is to simply add servers within one of our customer-oriented splits. Operationally, it should be relatively simple to add a cloned system to our service for any given customer, assuming that we do not have a great deal of state enabled. Therefore, in an effort to reduce the overall cost of scale, we will probably implement a z-axis split with an x-axis split in each z-axis segment. We can also now scale horizontally via x-axis replication within each of our $N$ number of z-axis pods. If a customer grows significantly in terms of the volume of its transactions, we can perform a cost-effective x-axis split (the addition of more cloned servers or virtual machines) within that customer’s pod.

Of course, as we have previously mentioned, the z-axis split really does not help us with code complexity. As our functionality increases and the size of our application grows, performing x- and z-axis splits alone will not allow us to focus and gain experience on specific features or services. Our time to market will likely suffer as the monolithic application grows in complexity. We may also find that the large monolithic z- and x-axis splits will not help us enough given all of the functions that need cached data. A single, very active customer,
focused on many of its own clients within our application, may find that a monolithic application is just too slow. This scenario would force us to focus on y-axis splits as well.

The y-axis split has its own set of problems when implemented in isolation. The first is similar to the problem of the x-axis split, in that a single server focused on a subset of functionality results in the functionality being unavailable when the server fails. As with the z-axis split, we will want to increase our availability by adding another cloned or x-axis server for each of our functions. We can save money by adding servers in an x-axis fashion for each of our y-axis splits versus continuing to split along the y-axis. Rather than modifying the code and further deconstructing it, we simply add servers into each of our y-axis splits and bypass the cost of further code modification.

The y-axis split also does not scale as well with customer growth as the z-axis split does. The y-axis splits focus more on the cache-ability of similar functions and work well when we have an application growing in size and complexity. Imagine, however, that you have decided to perform a y-axis split of your login functionality and that many of your client logins happen between 6 a.m. and 9 a.m. Pacific Time. Assuming that you need to cache data to allow for efficient logins, you will likely find that you need to perform a z-axis split of the login process to gain a higher cache hit ratio. As stated earlier, y-axis splits help most when you face a scenario of growth in the application and functionality, x-axis splits are most cost-effective when you must deal with transaction growth, and z-axis splits aid most when your organization is experiencing growth in the number of customers and users.

As we’ve stated previously, the x-axis approach is often the easiest to implement and, as such, is typically the very first type of split applied within systems or applications. It scales well with transaction volume, assuming that the application does not grow in complexity and that the transactions come from a defined base of slowly growing customers. As your product becomes more feature rich, you are forced to start looking at ways to make the system respond more quickly to user requests. You do not want, for instance, long searches to slow down the average response time of short-duration activities such as logins. To resolve average response time issues caused by competing functions, you need to implement a y-axis split.

Splitting along the x-axis is not an elegant approach to scaling as your customer base grows. As the number of your customers increases and as the data elements necessary to support them within an application increases, you need to find ways to segment these data elements to allow for maximum cost-effective
AKF Application Scale Cube Summary

Here is a summary of the three axes of scale:

- The x-axis represents the distribution of the same work or mirroring of an application across multiple entities. It is useful for scaling transaction volume cost-effectively, but does not scale well with data volume growth.

- The y-axis represents the distribution and separation of work responsibilities by “verb” or action across multiple entities. The y-axis can improve development time, as services are now implemented separately. It also helps with transaction growth and fault isolation. It helps to scale data specific to features and functions, but does not greatly benefit an organization that is experiencing customer data growth.

- The z-axis represents distribution and segmentation of work by customer, customer need, location, or value. It can create fault isolation and scale along customer boundaries. It does not aid in the scenario of growth of data specific to features or functions, nor does it aid in reducing time to market.

Hence, x-axis splits are mirror images of functions, y-axis splits separate applications based on the work performed, and z-axis splits separate work by customer, location, or some value-specific identifier (e.g., a hash or modulus).

Practical Use of the Application Cube

If you know anything about airline reservations, you most likely learned about it in a systems class that discussed the SABRE (Semi-automated Business Research Environment) reservation system. American Airlines implemented the SABRE system to automate reservations. IBM developed SABRE in the late 1950s and ran it on two IBM 7090 mainframes. This type of mainframe system was the prototypical monolithic system that relied on very large compute infrastructure to process high volumes of transactions and maintain a very high availability with built-in hardware redundancy. Fast-forward to today, and the airline reservation and pricing systems look very different.

Today’s airline reservation systems have to handle incredibly high volumes of
transactions. Web interfaces have allowed consumers to shop rapidly for multiple connection paths, travel times, and prices. The ratio known as “look-to-book” (how many flights a consumer looks at before booking one) is, on average, 100 to 1. Instead of relying on large compute platforms such as mainframes, some airlines have implemented software that uses all three axes of scale to provide super processing capability and high availability. One such software system is produced by PROS Holdings, Inc. (NYSE: PRO). PROS is a big data software company that helps its customers use big data to sell more seats effectively.

Before we explain how the PROS system is implemented, we need to delve into the very complex and sophisticated world of airline reservations and pricing. Our discussion here is by no means a thorough or exact explanation, but rather a simplification of the process to help you understand how the PROS system works.

Airline pricing is determined by a combination of available inventory and an origin/destination (O/D) model. The O/D model is used to understand air travelers’ true origins and destinations for any specific airport. For example, on a flight from Chicago O’Hare International Airport (ORD) to Los Angeles International Airport (LAX), there will be numerous passengers with many different origins and destinations. One passenger may only be traveling directly from Chicago to Los Angeles, another might be connecting in Los Angeles to Honolulu (HNL), while a third may have started in Newark, New Jersey (EWR), and is connecting on to San Francisco (SFO). In this case, the ORD → LAX flight serves the demand for at least three different O/Ds: ORD–LAX, ORD–HNL, and EWR–SFO. With an average seating capacity exceeding 150 passengers, there may be as many as 150 or more O/Ds. Measuring only the volume of passengers traveling the ORD–LAX route would overstate this demand while not reflecting the demand for the other routes. An O/D model estimates the true travel volume by airport pair based on the passengers’ entire journey and allows airlines to better understand and price flights for their origin and destination markets.

One of the PROS software systems provides this type of O/D model for airlines. Another PROS system uses this O/D model as input, along with an airline’s available inventory, to provide real-time dynamic pricing (RTDP). RTDP systems allow airlines to provide prices based on real-time demand and inventory to customers. This is not done directly; rather, customers make requests and receive responses through global distribution systems (GDS). These GDS are used by online travel agents, including an airline’s own Web site, to
aggregate pricing data. Examples of GDS include Sabre (which owns and powers Travelocity), Amadeus, and Travelport.

Now that we know a little bit about airline pricing and reservations, we can look at how PROS engineers architected the PROS product for high availability and scalability. **Figure 23.3** shows a typical implementation.

As shown in **Figure 23.3**, the O/D model is provided by a service completely separated from the real-time dynamic pricing service and distributed asynchronously by the cache distributor. This is a y-axis separation that provides fault isolation. Should the O/D model service fail, the RTDP service can continue to provide pricing, albeit with a slightly out-of-date demand model.

In our depiction of the system, there are three implementations of the RTDP service, each providing dynamic pricing to a different GDS. Thus each GDS and
RTDP pair together form an isolated “swim lane” of functionality. This again allows for fault isolation. While highly unlikely, it is possible that a GDS will make such a high volume of requests that it will slow down the RTDP service. Should this occur, segmentation of the GDS and pairing of a GDS with a single RTDP keeps other GDS instances from being affected. Additionally, the volumes of pricing requests vary greatly depending on the GDS: Some provide several months’ worth of options at once, whereas others request days’ or weeks’ worth of information. This z-axis segmentation allows the RTDP system to scale independently based on the GDS needs. Each of the services—RTDP, O/D, the inventory stream from the airline—as well as the DB cluster have multiple instances of the respective software on different servers. This is an x-axis split that ensures there are no single points of failure.

Observations

Which split is appropriate for your product? How many splits should you incorporate in your product? The answers to these questions aren’t always straightforward and easy to find. In the absence of data, the best approach is to gather the engineering team and review the architecture for likely scale bottlenecks. Over time, as data is collected and evaluated, the answer will become clearer about the best way to approach scaling the product.

Where to draw the line with y-axis splits is not always easy to decide. If you have tens of thousands of features or “verbs,” it doesn’t make sense to have tens of thousands of splits. You want to have manageable sizes of code bases in each of the splits, but not so many splits that the absolute number itself becomes unmanageable. You also want the cache sizes in your production environment to be manageable. Both of these factors should be considered when you are determining where you should perform splits and how many you should have.

In general, z-axis splits are a little easier from a design perspective. Ideally, you will simply design a system that has flexibility built into it. We previously mentioned a configurable number $N$ in both the ecommerce and back-office IT systems. This number allows us to start splitting application flows by customer within the system. As our business grows, we simply increase $N$ to allow for greater segmentation and to help smooth the load across our production systems. Of course, potentially some work must be done in data storage (where those customers live), as we will discuss in Chapter 24, but we expect that you can develop tools to help manage that work. With the y-axis, unfortunately, it is not so easy to design flexibility into the system.

As always, the x-axis is relatively easy to split and handle because it is always
just a duplicate of its peers. In all of our previous cases, the $x$-axis was always subordinate to the $y$- and $z$-axes. This is almost always the case when you perform $y$- and $z$-axis splits. To the point, the $x$-axis becomes relevant within either a $y$- or $z$-axis split. Sometimes, the $y$- or $z$-axis, as was the case in more than one of the examples, is subordinate to the other, but in nearly all cases, the $x$-axis is subordinate to either the $y$- or $z$-axis whenever the $y$- or $z$-axis or both are employed.

What do you do if and when your business contracts? If you’ve split to allow for aggressive hyper-growth and the economy presents your business with a downward cycle not largely under your control, what do you do? The $x$-axis splits are easy to unwind: You simply remove the systems you do not need. If those systems are fully depreciated, you can simply power them off for future use when your business rebounds. The $y$-axis splits might be hosted on a smaller number of systems, potentially leveraging virtual machine software to carve a set of physical servers into multiple servers. The $z$-axis splits should also be capable of being collapsed onto similar systems either through the use of virtual machine software or just by changing the boundaries that indicate which customers reside on which systems.

**Conclusion**

This chapter discussed the employment of the AKF Scale Cube to applications within a product, service, or platform. We modified the AKF Scale Cube slightly, narrowing the scope and definition of each of the axes so that it became more meaningful to application and systems architecture and the production deployment of applications.

Our $x$-axis still addresses the growth in transactions or work performed by any platform or system. Although the $x$-axis handles growth in transaction volume well, it suffers when application complexity increases significantly (as measured through the growth in functions and features) or when the number of customers with cacheable data needs grows significantly.

The $y$-axis addresses application complexity and growth. As we grow our product to become more feature rich, it requires more resources. Furthermore, transactions that would otherwise complete quickly start to slow down as demand-laden systems mix both fast and slow transactions. In such a scenario, our ability to cache data for all features starts to drop as we run into system constraints. The $y$-axis helps address all of these conditions while simultaneously benefiting our production teams. Engineering teams can focus on smaller portions of our more complex code base. As a result, defect rates decrease, new
engineers get up to speed faster, and expert engineers can develop software faster. Because all axes address transaction scale as well, the y-axis also benefits us as we grow the transactions against our system, but it is not as easily scaled in this dimension as the x-axis.

The z-axis addresses growth in customer base. As we will see in Chapter 24, it can also help us address growth in other data elements, such as product catalogs. As transactions and customers grow, and potentially as transactions per customer grow, we might find ourselves in a position where we might need to address the specific needs of a class of customer. This need might arise solely because each customer has an equal need for some small cache space, but it might be the case that the elements you cache by customer are distinct based on some predefined customer class. Either way, segmenting by requestor, customer, or client helps solve that problem. It also helps us scale along the transaction growth path, albeit not as easily as with the x-axis.

As indicated in Chapter 22, not all companies need all three axes of scale to survive. When more than one axis is employed, the x-axis is almost always subordinate to the other axes. You might, for instance, have multiple x-axis splits, each occurring within a y- or z-axis split. When employing y- and z-axis splits together (typically with an x-axis split), either split can become the “primary” means of splitting. If you split first by customer, you can still make y-axis functionality implementations within each of your z-axis splits. These would be clones of each other such that the login service in z-axis customer split 1 looks exactly like the login service for z-axis customer split N. The same is true for a y-axis primary split: The z-axis implementations within each functionality split would be similar or clones of each other.

**Key Points**

- The x-axis application splits scale linearly with transaction growth. They do not help with the growth in code complexity, customers, or data, however. The x-axis splits are “clones” of each other.
- The x-axis tends to be the least costly to implement, but suffers from constraints in instruction size and data set size.
- The y-axis application splits help scale code complexity as well as transaction growth. They are mostly meant for code scale, as they are not as efficient as x-axis splits for handling transaction growth.
- The y-axis application splits also aid in reducing cache sizes where cache sizes scale with function growth.
• In general, y-axis splits tend to be more costly to implement than x-axis splits as a result of the more extensive engineering time needed to separate monolithic code bases.

• The y-axis splits aid in fault isolation.

• Although y-axis splits can be performed without code modification, you might not get the benefit of cache size reduction and you will not get the benefit of decreasing code complexity.

• The y-axis splits can help scale organizations by reducing monolithic code complexity.

• The z-axis application splits help scale customer growth, some elements of data growth (as we will see in Chapter 24), and transaction growth.

• The z-axis application splits can help reduce cache sizes where caches scale in relation to the growth in users or other data elements.

• Like y-axis splits, z-axis splits aid in fault isolation. They, too, can be implemented without code changes but may not realize the benefit of cache size reduction without some code modification.

• The z-axis splits can aid with incident impact reduction and decrease the risk associated with deployments in either phased or continuous delivery environments.

• The z-axis splits can reduce customer response times both by reducing data set size and allowing solutions to be located “close to the customer” geographically.

• The choice of when to use which method or axis of scale is both art and science. Intuition is typically the initial guiding force, whereas production data should be used over time to help inform the decision-making process.