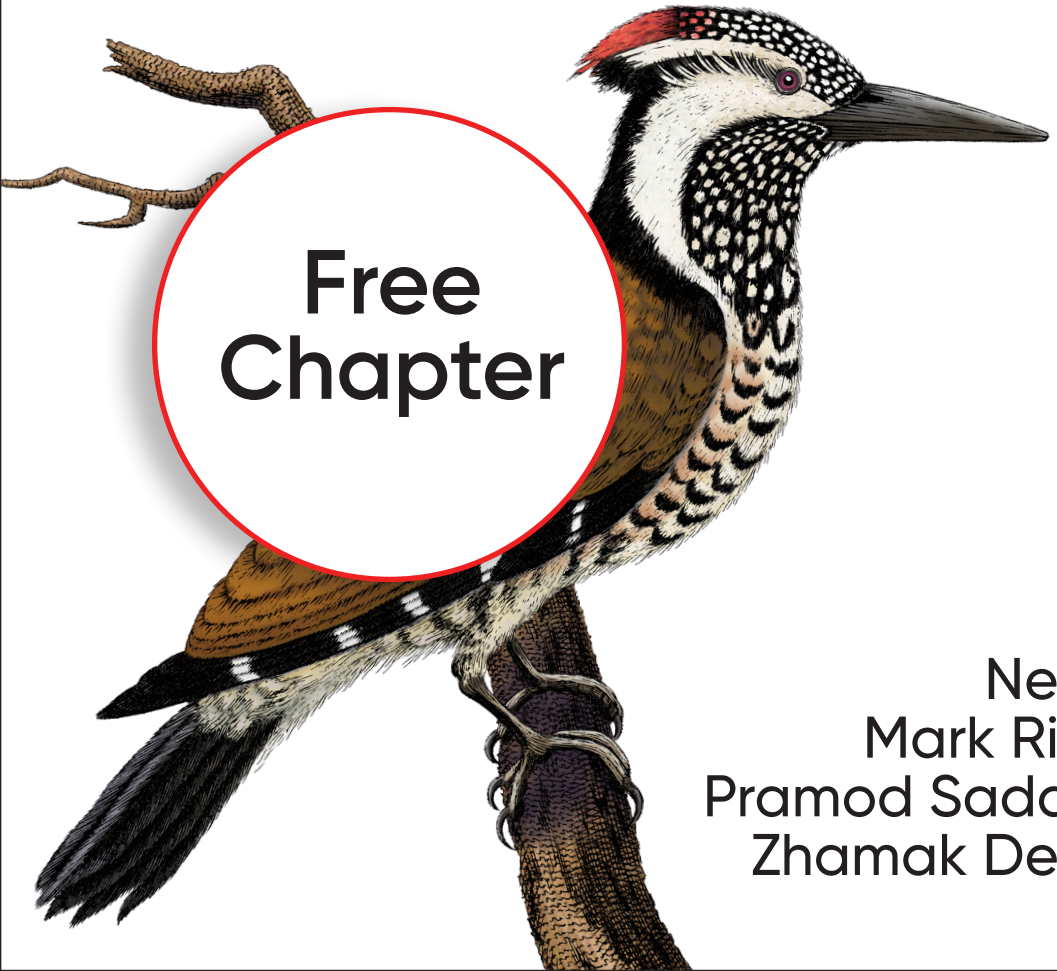
O'REILLY®

Software Architecture: The Hard Parts

Modern Trade-Off Analyses for Distributed Architectures



Free Chapter

Neal Ford,
Mark Richards,
Pramod Sadalage &
Zhamak Dehghani

Software Architecture: The Hard Parts

Modern Trade-Off Analysis for Distributed Architectures

This excerpt contains Chapter 7. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Software Architecture: The Hard Parts
by Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani

Printed in Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield
Development Editor: Nicole Taché
Production Editor: Christopher Faucher
Copyeditor: Sonia Saruba
Proofreader: Sharon Wilkey

Indexer: Sue Klefstad
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: O'Reilly Media, Inc.

October 2021: First Edition

Revision History for the First Edition
2021-09-23: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492086895* for release details.

978-1-492-08689-5

[MBP]

Table of Contents

CHAPTER 7
Service Granularity

`Thursday, October 14, 13:33`

As the migration effort got underway, both Addison and Austen started getting overwhelmed with all of the decisions involved with breaking apart the domain services previously identified. The development team also had its own opinions, which made decision making for service granularity even more difficult.

"I'm still not sure what to do with the core ticketing functionality," said Addison. "I can't decide whether ticket creation, completion, expert assignment, and expert routing should be one, two, three, or even four services. Taylen is insisting on making everything fine-grained, but I'm not sure that's the right approach."

"Me neither," said Austen. "And I've got my own issues trying to figure out if the customer registration, profile management, and billing functionality should even be broken apart. And on top of all that, I've got another game this evening."

"You've always got a game to go to," said Addison. "Speaking of customer functionality, did you ever figure out if the customer login functionality is going to be a separate service?"

"No," said Austen, "I'm still working on that as well. Skyler says it should be separate, but won't give me a reason other than to say it's separate functionality."

"This is hard stuff," said Addison. "Do you think Logan can shed any light on this?"

"Good idea," said Austen, "This seat-of-the-pants analysis is really slowing things down."

Addison and Austen invited Taylen, the Sysops Squad tech lead, to the meeting with Logan so that all of them could be on the same page with regard to the service granularity issues they were facing.

"I'm telling you," said Taylen, "we need to break up the domain services into smaller services. They are simply too coarse-grained for microservices. From what I remember, *micro* means small. We are, after all, moving to microservices. What Addison and Austen are suggesting simply doesn't fit with the microservices model."

"Not every portion of an application has to be microservices," said Logan. "That's one of the biggest pitfalls of the microservices architecture style."

"If that's the case, then how do you determine what services should and shouldn't be broken apart?" asked Taylen.

"Let me ask you something, Taylen," said Logan. "What is your reason for wanting to make all of the services so small?"

"Single-responsibility principle," answered Taylen. "Look it up. That's what microservices is based on."

"I know what the single-responsibility principle is," said Logan. "And I also know how subjective it can be. Let's take our customer notification service as an example. We can notify our customers through SMS, email, and we even send out postal letters. So tell me everyone, one service or three services?"

"Three," immediately answered Taylen. "Each notification method is its own thing. That's what microservices is all about."

"One," answered Addison. "Notification itself is clearly a single responsibility."

"I'm not sure," answered Austen. "I can see it both ways. Should we just toss a coin?"

"This is exactly why we need help," sighed Addison.

"The key to getting service granularity right," said Logan, "is to remove opinion and gut feeling, and use granularity disintegrators and integrators to objectively analyze the trade-offs and form solid justifications for whether or not to break apart a service."

"What are granularity disintegrators and integrators?" asked Austen.

"Let me show you," said Logan.

Architects and developers frequently confuse the terms *modularity* and *granularity*, and in some cases even treat them to mean the same thing. Consider the following dictionary definitions of each of these terms:

Modularity
: Constructed with standardized units or dimensions for flexibility and variety in use.

Granularity
: Consisting of or appearing to consist of one of numerous particles forming a larger unit.

It's no wonder so much confusion exists between these terms! Although the terms have similar dictionary definitions, we want to distinguish between them because they mean different things within the context of software architecture. In our usage, *modularity* concerns breaking up systems into separate parts (see Chapter 3), whereas *granularity* deals with the *size* of those separate parts. Interestingly enough, most issues and challenges within distributed systems are typically not related to modularity, but rather granularity.

Determining the right level of granularity—the size of a service—is one of the many hard parts of software architecture that architects and development teams continually struggle with. Granularity is not defined by the number of classes or lines of code in a service, but rather what the service does—hence why it is so hard to get service granularity right.

Architects can leverage metrics to monitor and measure various aspects of a service to determine the appropriate level of service granularity. One such metric used to objectively measure the size of a service is to calculate the number of statements in a service. Every developer has a different coding style and technique, which is why the number of classes and number of lines of code are poor metrics to use to measure granularity. The number of statements, on the other hand, at least allows an architect or development team to objectively measure *what* the service is doing. Recall from Chapter 4 that a *statement* is a single complete action performed in the source code, usually terminated by a special character (such as a semicolon in languages such as Java, C, C++, C#, Go, JavaScript; or a newline in languages such as F#, Python, and Ruby).

Another metric to determine service granularity is to measure and track the number of *public* interfaces or operations exposed by a service. Granted, while there is still a bit of subjectiveness and variability with these two metrics, it's the closest thing we've come up with so far to objectively measure and assess service granularity.

Two opposing forces for service granularity are granularity disintegrators and granularity integrators. These opposing forces are illustrated in Figure 7-1. *Granularity disintegrators* address the question "When should I consider breaking apart a service into smaller parts?", whereas *Granularity integrators* address the question "When should I consider putting services back together?" One common mistake many development teams make is focusing too much on granularity disintegrators while ignoring granularity integrators. The secret of arriving at the appropriate level of granularity for a service is achieving an equilibrium between these two opposing forces.

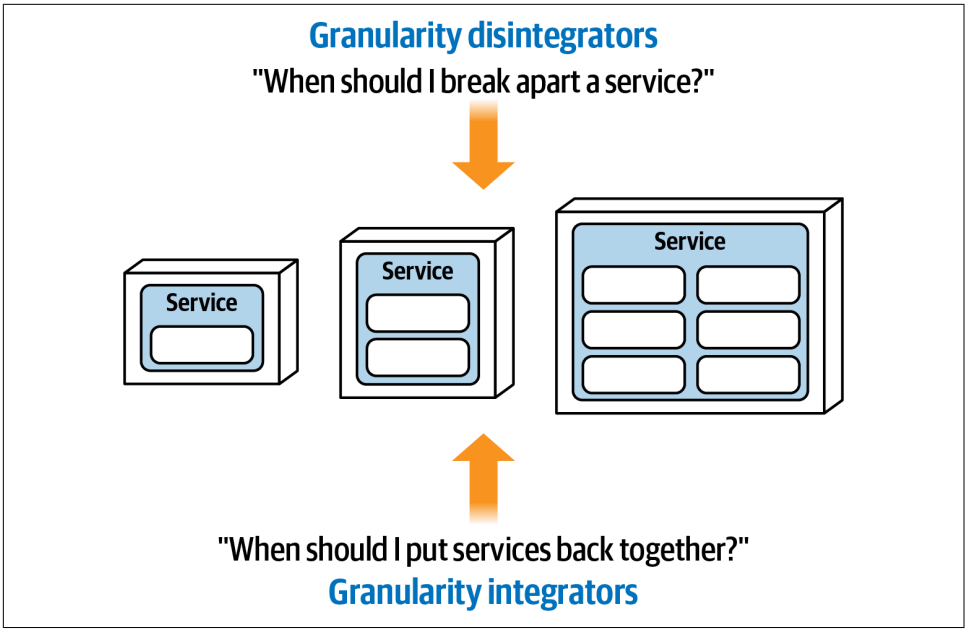


Figure 7-1. Service granularity depends on a balance of disintegrators and integrators

Granularity Disintegrators

Granularity disintegrators provide guidance and justification for when to break a service into smaller pieces. While the justification for breaking up a service may involve only a single driver, in most cases the justification will be based on multiple drivers. The six main drivers for granularity disintegration are as follows:

Service scope and function
: Is the service doing too many unrelated things?

Code volatility
: Are changes isolated to only one part of the service?

Scalability and throughput
: Do parts of the service need to scale differently?

Fault tolerance
: Are there errors that cause critical functions to fail within the service?

Security
: Do some parts of the service need higher security levels than others?

Extensibility
: Is the service always expanding to add new contexts?

The following sections detail each of these granularity disintegration drivers.

Service Scope and Function

The service scope and function is the first and most common driver for breaking up a single service into smaller ones, particularly with regard to microservices. There are two dimensions to consider when analyzing the service scope and function. The first dimension is *cohesion*: the degree and manner to which the operations of a particular service interrelate. The second dimension is the overall *size* of a component, measured usually in terms of the total number of statements summed from the classes that make up that service, the number of public entrypoints into the service, or both.

Consider a typical Notification Service that does three things: notifies a customer through SMS (Short Message Service), email, or a printed postal letter that is mailed to the customer. Although it is very tempting to break this service into three separate single-purpose services (one for SMS, one for email, and one for postal letters) as illustrated in Figure 7-2, this alone is not enough to justify breaking the service apart because it already has relatively strong cohesion—all of these functions relate to one thing, notifying the customer. Because "single purpose" is left for individual opinion and interpretation, it is difficult to know whether to break apart this service or not.

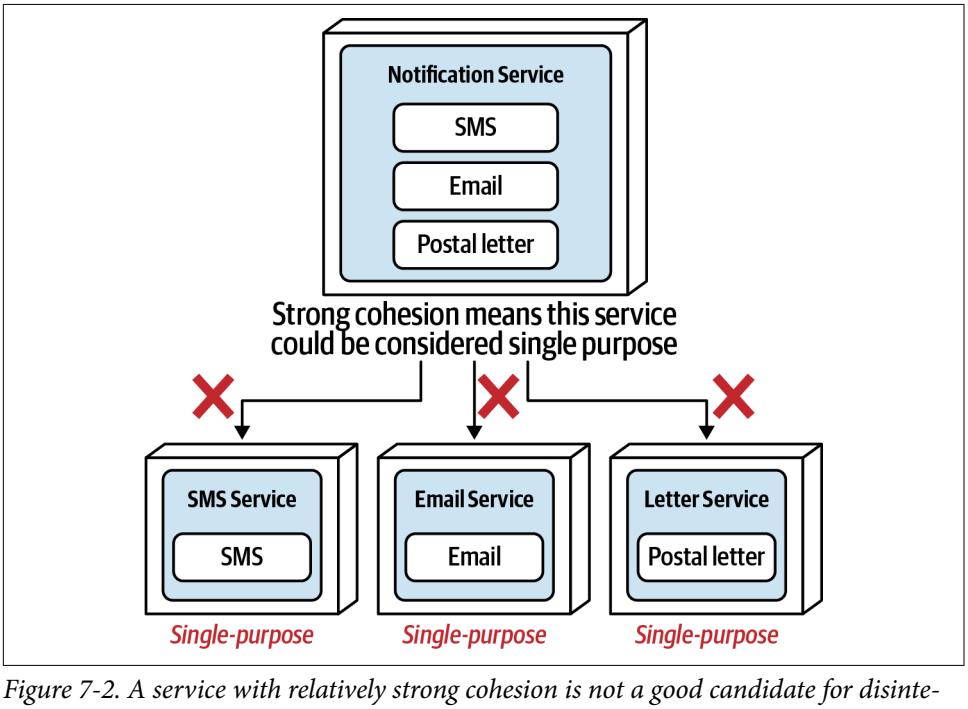


Figure 7-2. A service with relatively strong cohesion is not a good candidate for disintegration based on functionality alone

Now consider a single service that manages the customer profile information, customer preferences, and also customer comments made on the website. Unlike the previous Notification Service example, this service has relatively weak cohesion because these three functions relate to a broader scope—customer. This service is possibly doing too much, and hence should probably be broken into three separate services, as illustrated in Figure 7-3.

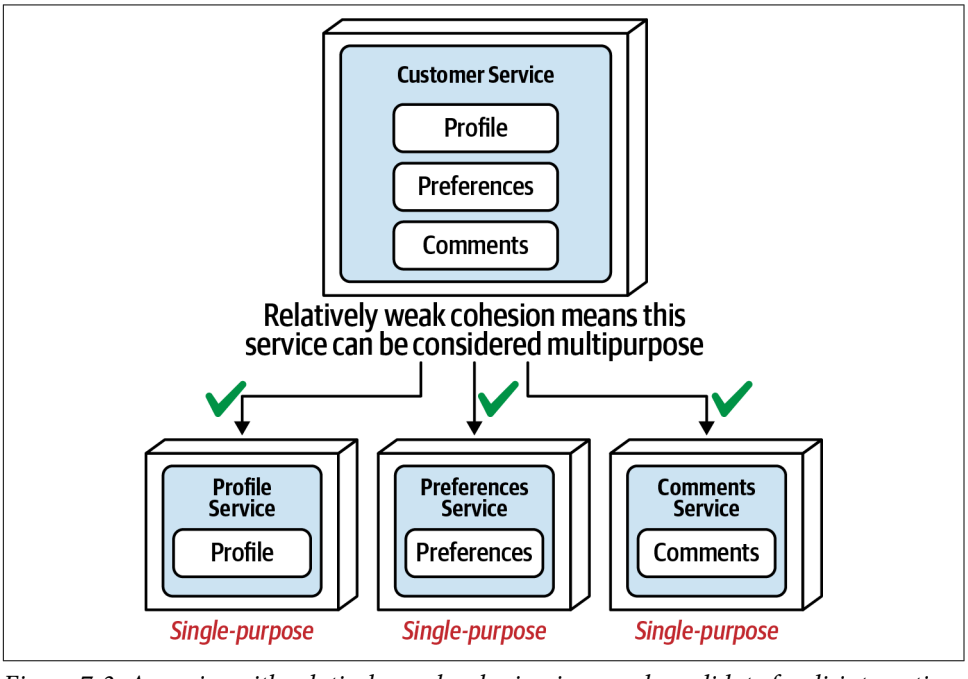


Figure 7-3. A service with relatively weak cohesion is a good candidate for disintegration

This granularity disintegrator is related to the single-responsibility principle coined by Robert C. Martin as part of his SOLID principles, which states, "every class should have responsibility over a single part of that program's functionality, which it should encapsulate. All of that module, class or function's services should be narrowly aligned with that responsibility." While the single-responsibility principle was originally scoped within the context of classes, in later years it has expanded to include components and services.

Within the microservices architecture style, a *microservice* is defined as a single-purpose, separately deployed unit of software that does *one thing* really well. No wonder developers are so tempted to make services as small as possible without considering why they are doing so! The subjectiveness related to what is and isn't a single responsibility is where most developers get into trouble with regard to service granularity. While there are some metrics (such as LCOM) to measure cohesion, it is nevertheless highly subjective when it comes to services—is notifying the customer one single thing, or is notifying via email one single thing? For this reason, it is vital to understand other granularity disintegrators to determine the appropriate level of granularity.

Code Volatility

Code volatility--the rate at which the source code changes—is another good driver for breaking a service into smaller ones. This is also known as *volatility-based decomposition*. Objectively measuring the frequency of code changes in a service (easily done through standard facilities in any source code version-control system) can sometimes lead to a good justification for breaking apart a service. Consider the notification service example again from the prior section. Service scope (cohesion) alone was not enough to justify breaking the service apart. However, by applying change metrics, relevant information is revealed about the service:

- SMS notification functionality rate of change: every six months (avg)
- Email notification functionality rate of change: every six months (avg)
- Postal letter notification functionality rate of change: weekly (avg)

Notice that the postal letter functionality changes weekly (on average), whereas the SMS and email functionality rarely changes. As a single service, any change to the postal letter code would require the developer to test and redeploy the entire service, including SMS and email functionality. Depending on the deployment environment, this also might mean SMS and email functionality would not be available when the postal letter changes are deployed. Thus, as a single service, testing scope is increased and deployment risk is high. However, by breaking this service into two separate services (Electronic Notification and Postal Letter Notification), as illustrated in Figure 7-4, frequent changes are now isolated into a single, smaller service. This in turn means that the testing scope is significantly reduced, deployment risk is lower, and SMS and email functionality is not disrupted during a deployment of postal letter changes.

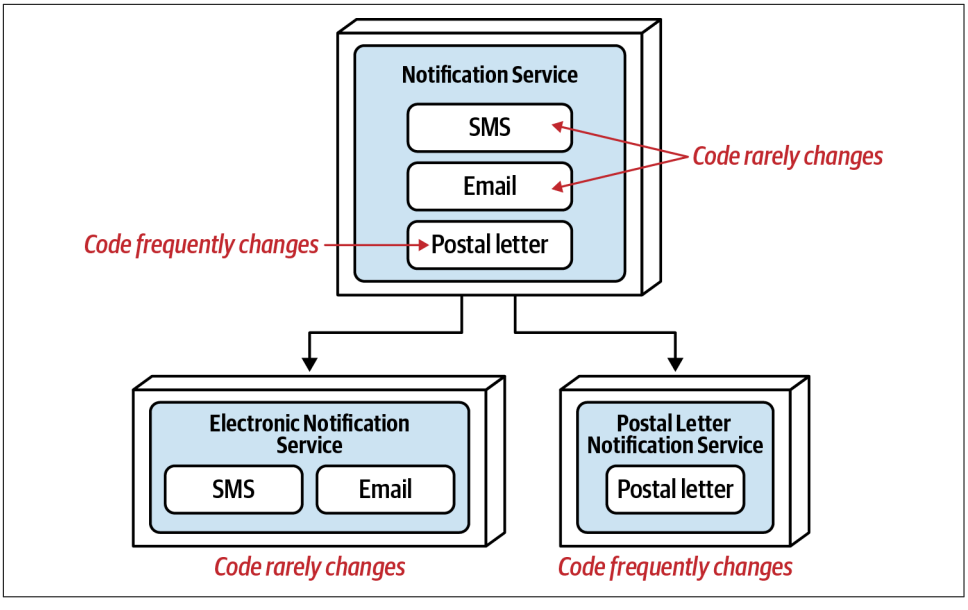


Figure 7-4. An area of high code change in a service is a good candidate for disintegration

Scalability and Throughput

Another driver for breaking up a service into separate smaller ones is *scalability* and *throughput*. The scalability demands of different functions of a service can be objectively measured to qualify whether a service should be broken apart. Consider once again the Notification Service example, where a single service notifies customers through SMS, email, and printed postal letter. Measuring the scalability demands of this single service reveals the following information:

- SMS notification: 220,000/minute
- Email notification: 500/minute
- Postal letter notification: 1/minute

Notice the extreme variation between sending out SMS notifications and postal letter notifications. As a single service, email and postal letter functionality must unnecessarily scale to meet the demands of SMS notifications, impacting cost and also elasticity in terms of mean time to startup (MTTS). Breaking the Notification Service into three separate services (SMS, Email, and Letter), as illustrated in Figure 7-5, allows each of these services to scale independently to meet their varying demands of throughput.

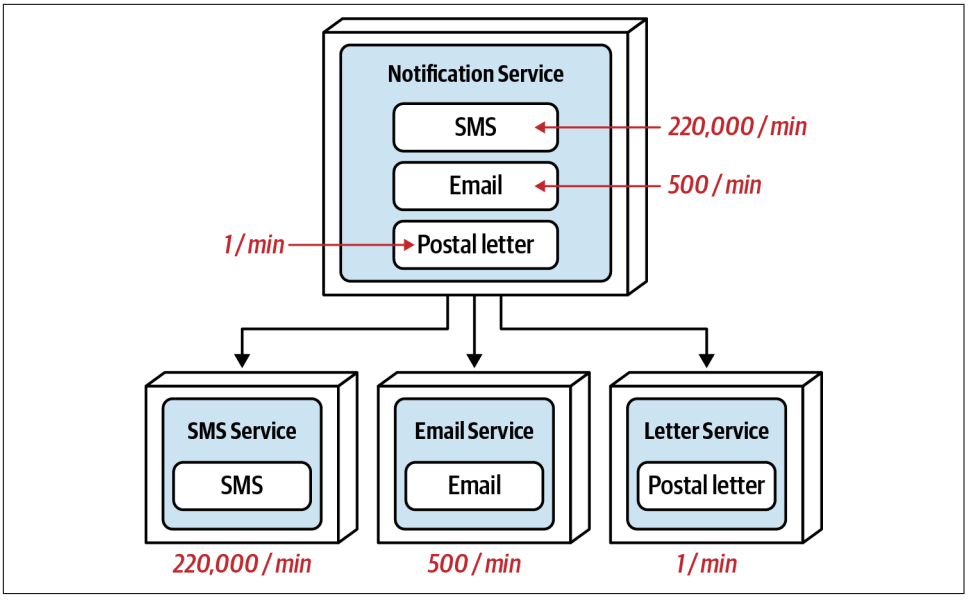


Figure 7-5. Differing scalability and throughput needs is a good disintegration driver

Fault Tolerance

Fault tolerance describes the ability of an application or functionality within a particular domain to continue to operate, even though a fatal crash occurs (such as an out-of-memory condition). Fault Tolerance is another good driver for granularity disintegration.

Consider the same consolidated Notification Service example that notifies customers through SMS, email, and postal letter (Figure 7-6). If the email functionality continues to have problems with out-of-memory conditions and fatally crashes, the entire service comes down, including SMS and postal letter processing.

Separating this single consolidated Notification Service into three separate services provides a level of fault tolerance for the domain of customer notification. Now, a fatal error in the functionality of the email service doesn't impact SMS or postal letters.

Notice in this example that the Notification Service is split into three separate services (SMS, Email, and Postal Letter), even though email functionality is the only issue with regard to frequent crashes (the other two are very stable). Since email functionality is the only issue, why not combine the SMS and postal letter functionality into a single service?

Consider the code volatility example from the prior section. In this case Postal Letter changes constantly, whereas the other two (SMS and Email) do not. Splitting this

service into only two services made sense because Postal Letter was the offending functionality, but Email and SMS are *related*—they both have to do with *electronically* notifying the customer. Now consider the fault-tolerance example. What do SMS notification and Postal Letter notification have in common other than a notification means to the customer? What would be an appropriate self-descriptive name of that combined service?

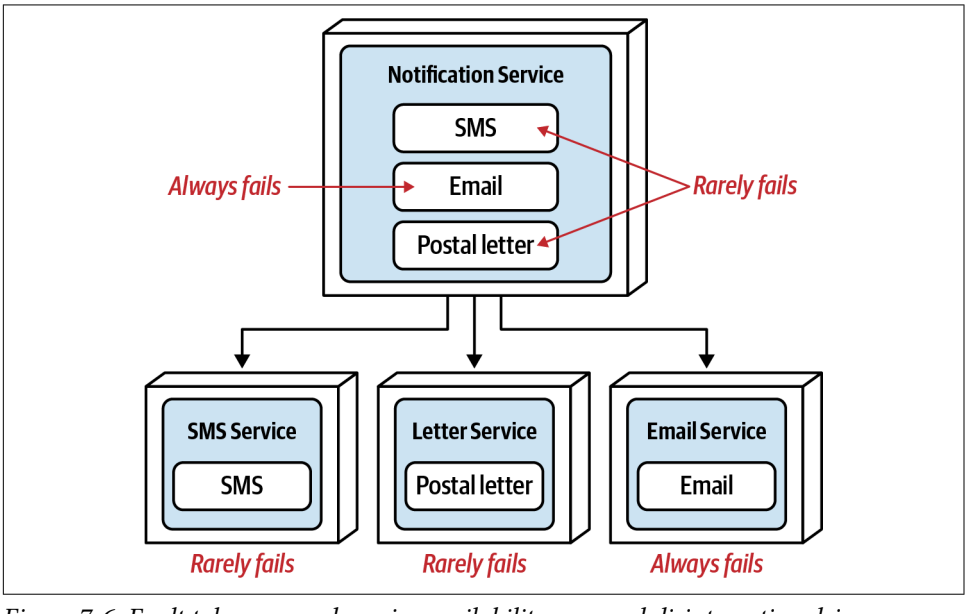


Figure 7-6. Fault tolerance and service availability are good disintegration drivers

Moving the email functionality to a separate service disrupts the overall *domain cohesion* because the resulting cohesion between SMS and postal letter functionality is weak. Consider what the likely service names would be: Email Service and…Other Notification Service? Email Service and…SMS-Letter Notification Service? Email Service and…Non-Email Service? This naming problem relates back to the service scope and function granularity disintegrator—if a service is too hard to name because it's doing multiple things, then consider breaking apart the service. The following disintegrations help in visualizing this important point:

- Notification Service → Email Service, Other Notification Service (poor name)
- Notification Service → Email Service, Non-Email Service (poor name)
- Notification Service → Email Service, SMS-Letter Service (poor name)
- Notification Service → Email Service, SMS Service, Letter Service (good names)

In this example, only the last disintegration makes sense, particularly considering the addition of another social media notification—where would that go? Whenever breaking apart a service, regardless of the disintegration driver, always check to see if strong cohesion can be formed with the "leftover" functionality.

Security

A common pitfall when securing sensitive data is to think only in terms of the storage of that data. For example, securing PCI (Payment Card Industry) data from non-PCI data might be addressed through separate schemas or databases residing in different secure regions. What is sometimes missing from this practice, however, is also securing *how* that data is accessed.

Consider the example illustrated in Figure 7-7 that describes a Customer Profile Service containing two main functions: customer profile maintenance for adding, changing, or deleting basic profile information (name, address, and so on); and customer credit card maintenance for adding, removing, and updating credit card information.

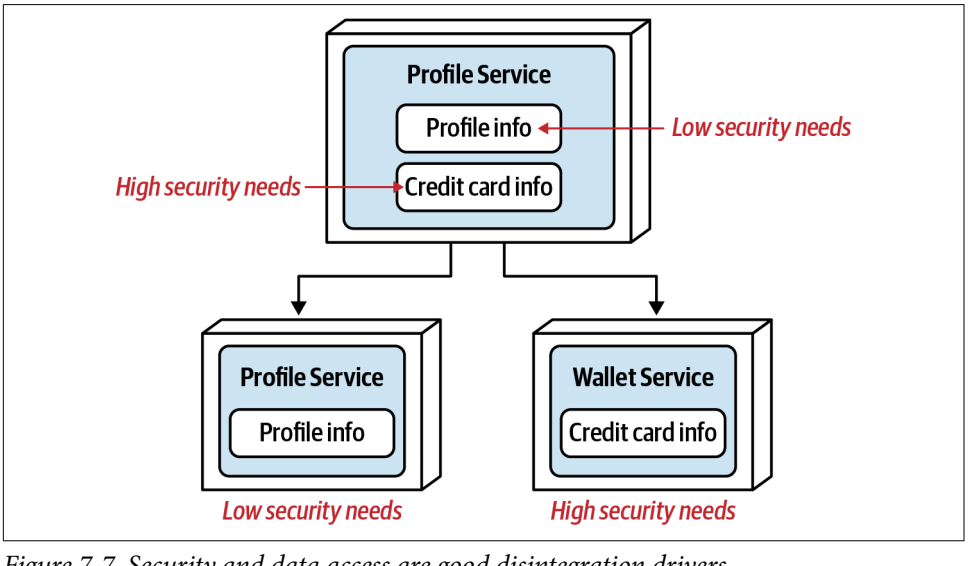


Figure 7-7. Security and data access are good disintegration drivers

While the credit card *data* may be protected, *access* to that data is at risk because the credit card functionality is joined together with the basic customer profile functionality. Although the API entry points into the consolidated customer profile service may differ, nevertheless there is risk that someone entering into the service to retrieve the customer name might also have access to credit card functionality. By breaking this service into two separate services, access to the *functionality* used to maintain credit

card information can be made more secure because the set of credit card operations is going into only a single-purpose service.

Extensibility

Another primary driver for granularity disintegration is_ extensibility_—the ability to add additional functionality as the service context grows. Consider a payment service that manages payments and refunds through multiple payment methods, including credit cards, gift cards, and PayPal transactions. Suppose the company wants to start supporting other managed payment methods, such as reward points, store credit from returns; and other third-party payment services, such as ApplePay, SamsungPay, and so on. How easy is it to extend the payment service to add these additional payment methods?

These additional payment methods could certainly be added to a single consolidated payment service. However, every time a new payment method is added, the entire payment service would need to be tested (including other payment types), and the functionality for all other payment methods unnecessarily redeployed into production. Thus, with the single consolidated payment service, the testing scope is increased and deployment risk is higher, making it more difficult to add additional payment types.

Now consider breaking up the existing consolidated service into three separate services (Credit Card Processing, Gift Card Processing, and PayPal Transaction Processing), as illustrated in Figure 7-8.

Now that the single payment service is broken into separate services by payment methods, adding another payment method (such as reward points) is only a matter of developing, testing, and deploying a single service separate from the others. As a result, development is faster, testing scope is reduced, and deployment risk is lower.

Our advice is to apply this driver only if it is known ahead of time that additional consolidated contextual functionality is planned, desired, or part of the normal domain. For example, with notification, it is doubtful the means of notification would continually expand beyond the basic notification means (SMS, email, or letter). However, with payment processing, it is highly likely that additional payment types would be added in the future, and therefore separate services for each payment type would be warranted. Since it is often difficult to sometimes "guess" whether (and when) contextual functionality might expand (such as additional payment methods), our advice is to wait on this driver as a primary means of justifying a granularly disintegration until a pattern can be established or confirmation of continued extensibility can be confirmed.

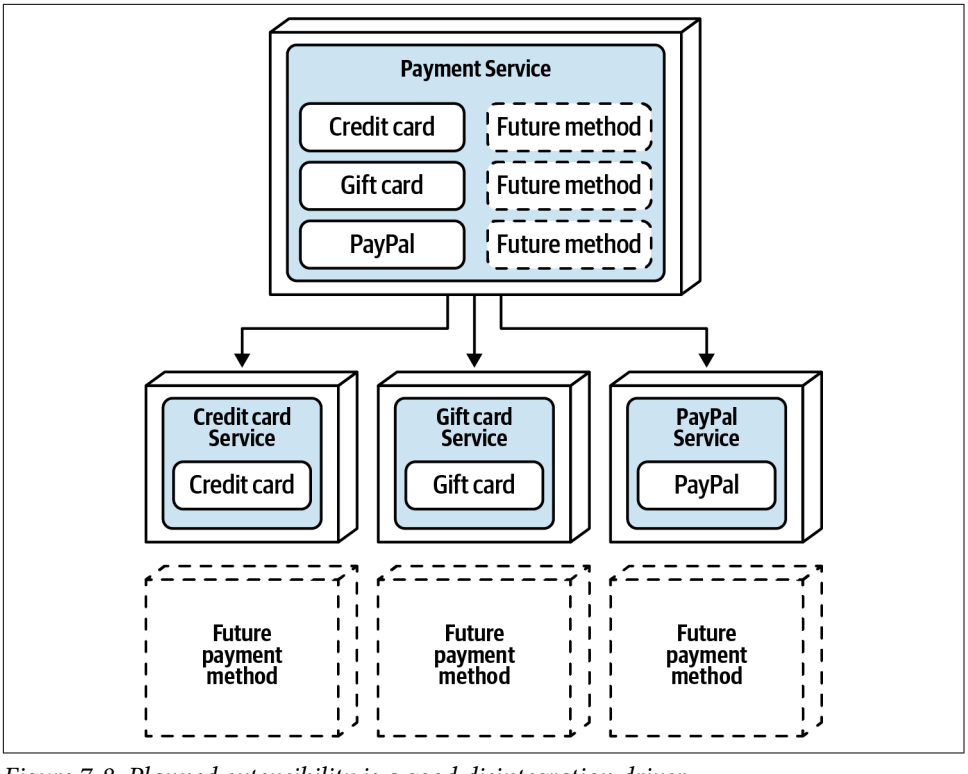


Figure 7-8. Planned extensibility is a good disintegration driver

Granularity Integrators

Whereas granularity disintegrators provide guidance and justification for when to break a service into smaller pieces, granularity integrators work in the opposite way—they provide guidance and justification for putting services back together (or not breaking apart a service in the first place). Analyzing the trade-offs between disintegration drivers and integration drivers is the secret to getting service granularity right. The four main drivers for granularity integration are as follows:

Database transactions
: Is an ACID transaction required between separate services?

Workflow and choreography
: Do services need to talk to one another? Shared code: Do services need to share codeamong one another? Database relationships: Although a service can be broken apart, can the data it uses be broken apart as well?

The following sections detail each of these granularity integration drivers.

Database Transactions

Most monolithic systems and course-grained domain services using relational databases rely on single-unit-of-work database transactions to maintain data integrity and consistency; see Chapter 9 for the details of ACID (database) transactions and how they differ from BASE (distributed) transactions. To understand how database transactions impact service granularity, consider the situation illustrated in Figure 7-9 where customer functionality has been split into a Customer Profile Service that maintains customer profile information and a Password Service that maintains password and other security-related information and functionality.

Notice that having two separate services provides a good level of security access control to password information since access is at a service level rather than at a request level. Access to operations such as changing a password, resetting a password, and accessing a customer's password for sign-in can all be restricted to a single service (and hence the access can be restricted to that single service). However, while this may be a good disintegration driver, consider the operation of registering a new customer, as illustrated in Figure 7-10.

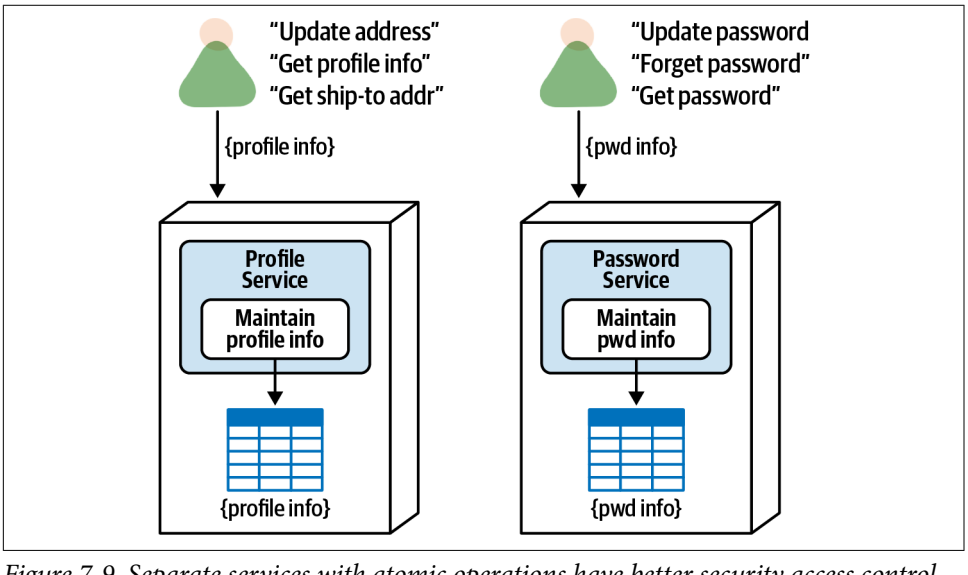


Figure 7-9. Separate services with atomic operations have better security access control

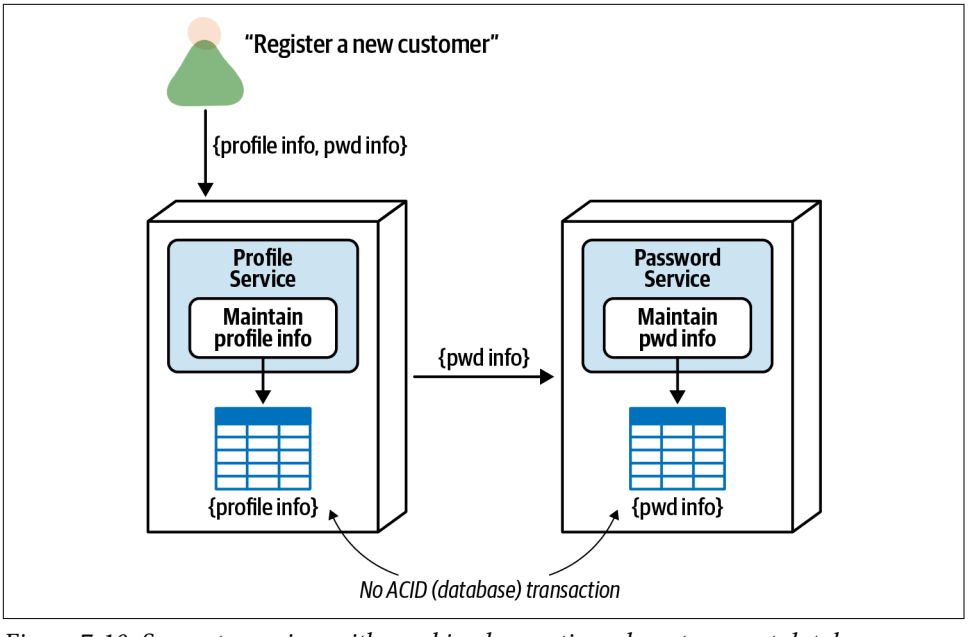


Figure 7-10. Separate services with combined operations do not support database (ACID) transactions

When registering a new customer, both profile and encrypted password information is passed into the Profile Service from a user interface screen. The Profile Service inserts the profile information into its corresponding database table, commits that work, and then passes the encrypted password information to the Password Service, which in turn inserts the password information into its corresponding database table and commits its own work.

While separating the services provides better security access control to the password information, the trade-off is that there is no ACID transaction for actions such as registering a new customer or unsubscribing (deleting) a customer from the system. If the password service fails during either of these operations, data is left in an inconsistent state, resulting in complex error handling (which is also error prone) to reverse the original profile insert or take other corrective action (see Chapter 12 for the details of eventual consistency and error handling within distributed transactions). Thus, if having a single-unit-of-work ACID transaction is required from a business perspective, these services should be consolidated into a single service, as illustrated in Figure 7-11.

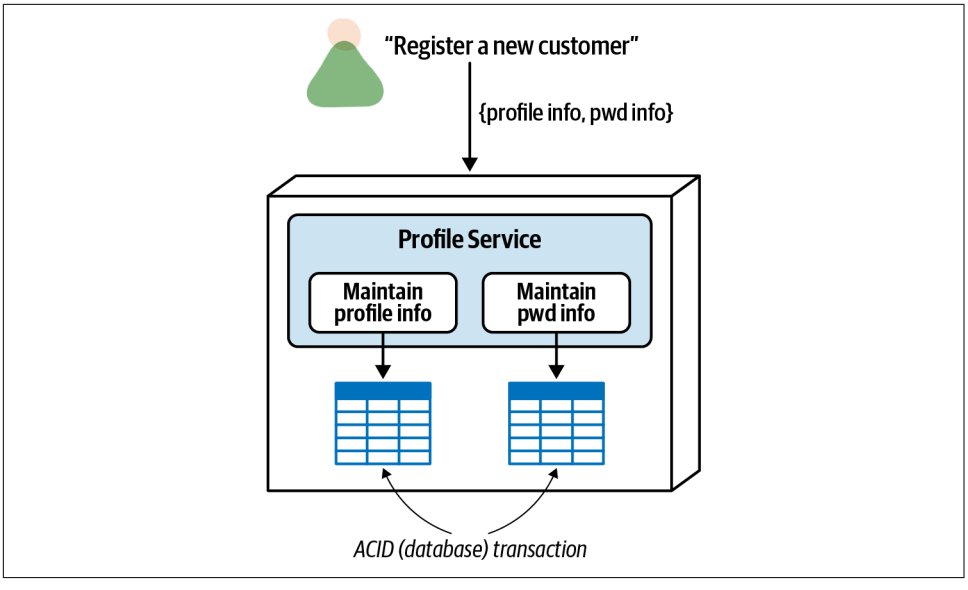


Figure 7-11. A single service supports database (ACID) transactions

Workflow and Choreography

Another common granularity integrator is *workflow* and *choreography*--services talking to one another (also sometimes referred to as *interservice* communication or *east-west* communications). Communication between services is fairly common and in many cases necessary in highly distributed architectures like microservices. However, as services move toward a finer level of granularity based on the disintegration factors outlined in the previous section, service communication can increase to a point where negative impacts start to occur.

Issues with overall fault tolerance is the first impact of too much synchronous inter-service communication. Consider the diagram in Figure 7-12: Service A communicates with services B and C, Service B communicates with Service C, Service D communicates with Service E, and finally Service E communicates with Service C. In this case, if Service C goes down, all other services become nonoperational because of a transitive dependency with Service C, creating an issue with overall fault tolerance, availability, and reliability.

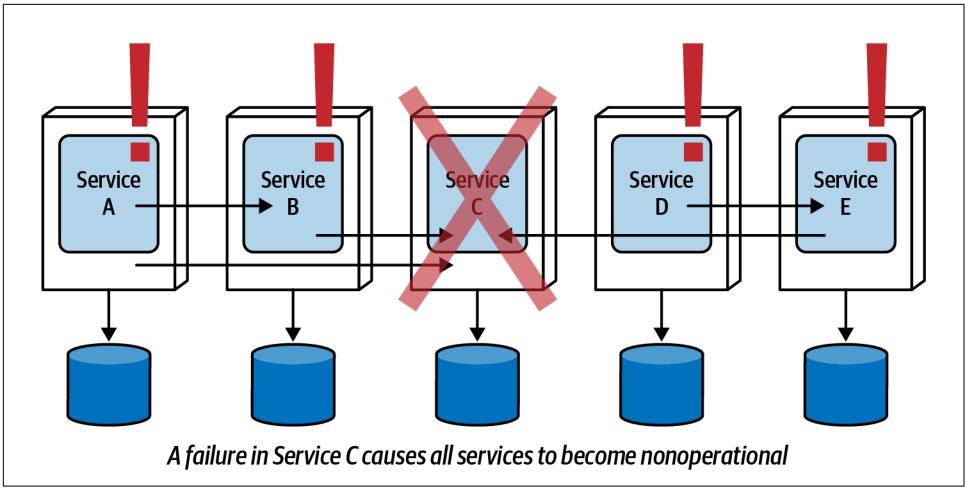


Figure 7-12. Too much workflow impacts fault tolerance

Interestingly enough, fault tolerance is one of the granularity disintegration drivers from the previous section—yet when those services need to talk to one another, nothing is really gained from a fault-tolerance perspective. When breaking apart services, always check to see if the functionalities are tightly coupled and dependent on one another. If it is, then overall fault tolerance from a business request standpoint won't be achieved, and it might be best to consider keeping the services together.

Overall performance and responsiveness is another driver for granularity integration (putting services back together). Consider the scenario in Figure 7-13: a large customer service is split into five separate services (services A through E). While each of these services has its own collection of cohesive atomic requests, retrieving all of the customer information collectively from a single API request into a single user interface screen involves five separate hops when using choreography (see Chapter 11 for an alternative solution to this problem using orchestration). Assuming 300 ms in network and security latency per request, this single request would incur an additional 1500 ms just in latency alone! Consolidating all of these services into a single service would remove the latency, therefore increasing overall performance and responsiveness.

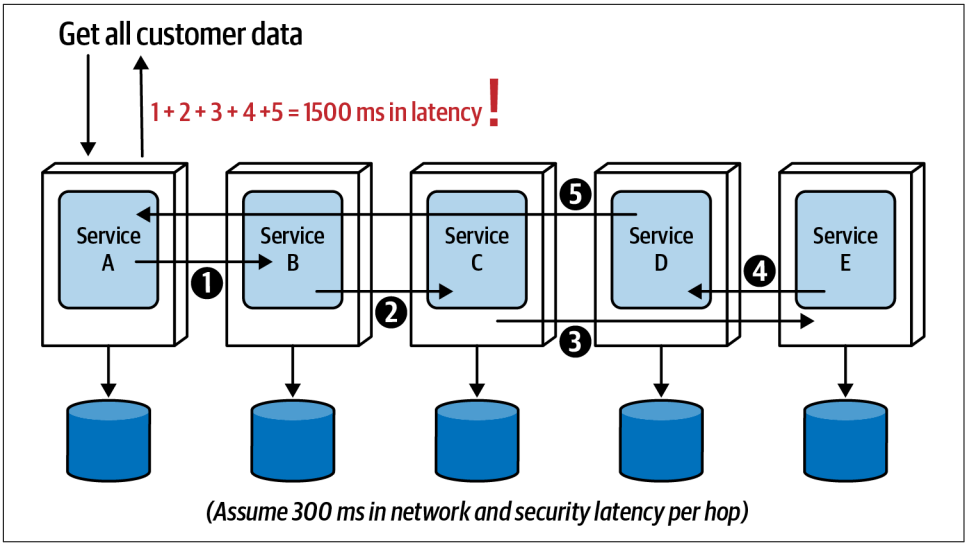


Figure 7-13. Too much workflow impacts overall performance and responsiveness

In terms of overall performance, the trade-off for this integration driver is balancing the need to break apart a service with the corresponding performance loss if those services need to communicate with one another. A good rule of thumb is to take into consideration the number of requests that require multiple services to communicate with one another, also taking into account the criticality of those requests requiring interservice communication. For example, if 30% of the requests require a workflow between services to complete the request and 70% are purely atomic (dedicated to only one service without the need for any additional communication), then it might be OK to keep the services separate. However, if the percentages are reversed, then consider putting them back together again. This assumes, of course, that overall performance matters. There's more leeway in the case of backend functionality where an end user isn't waiting for the request to complete.

The other performance consideration is with regard to the criticality of the request requiring workflow. Consider the previous example, where 30% of the requests require a workflow between services to complete the request, and 70% are purely atomic. If a critical request that requires extremely fast response time is part of that 30%, then it might be wise to put the services back together, even though 70% of the requests are purely atomic.

Overall reliability and data integrity are also impacted with increased service communication. Consider the example in Figure 7-14: customer information is separated into five separate customer services. In this case, adding a new customer to the system involves the coordination of all five customer services. However, as explained in a previous section, each of these services has its own database transaction. Notice in

Figure 7-14 that services A, B, and C have all committed part of the customer data, but Service D fails.

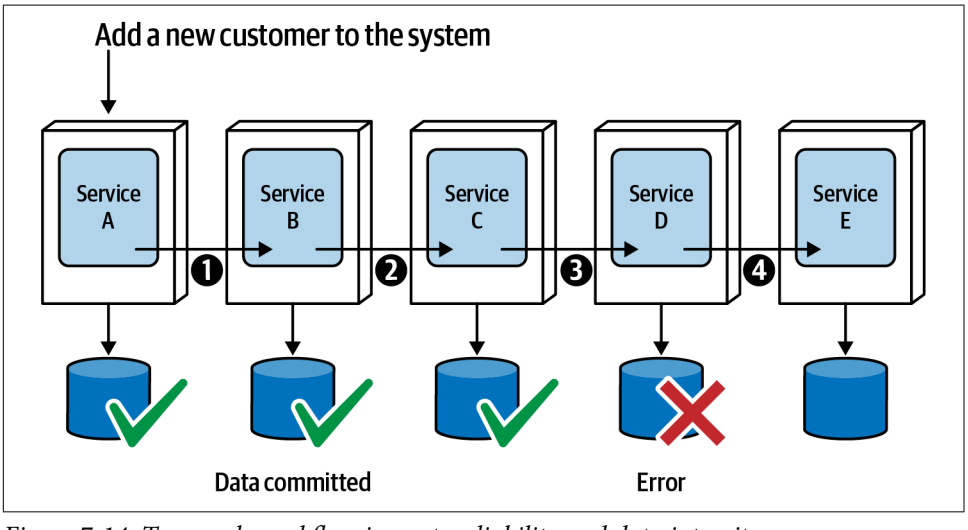


Figure 7-14. Too much workflow impacts reliability and data integrity

This creates a data consistency and data integrity issue because part of the customer data has already been committed, and may have already been acted upon through a retrieval of that information from another process or even a message sent out from one of those services broadcasting an action based on that data. In either case, that data would either have to be rolled back through compensating transactions or marked with a specific state to know where the transaction left off in order to restart it. This is very messy situation, one we describe in detail in Chapter 12. If data integrity and data consistency are important or critical to an operation, it might be wise to consider putting those services back together.

Shared Code

Shared source code is a common (and necessary) practice in software development. Functions like logging, security, utilities, formatters, converters, extractors, and so on are all good examples of shared code. However, things can get complicated when dealing with shared code in a distributed architecture and can sometimes influence service granularity.

Shared code is often contained in a shared library, such as a JAR file in the Java Ecosystem, a GEM in the Ruby environment, or a DLL in the .NET environment, and is typically bound to a service at compile time. While we dive into code reuse patterns in detail in Chapter 8, here we illustrate only how shared code can sometimes

influence service granularity and can become a granularity integrator (putting services back together).

Consider the set of five services shown in Figure 7-15. While there may have been a good disintegrator driver for breaking apart these services, they all share a common codebase of domain functionality (as opposed to common utilities or infrastructure functionality). If a change occurs in the shared library, this would eventually necessitate a change in the corresponding services using that shared library. We say *eventually* because versioning can sometimes be used with shared libraries to provide agility and backward compatibility (see Chapter 8). As such, all of these separately deployed services would have to be changed, tested, and deployed together. In these cases, it might be wise to consolidate these five services into a single service to avoid multiple deployments, as well as having the service functionality be out of sync based on the use of different versions of a library.

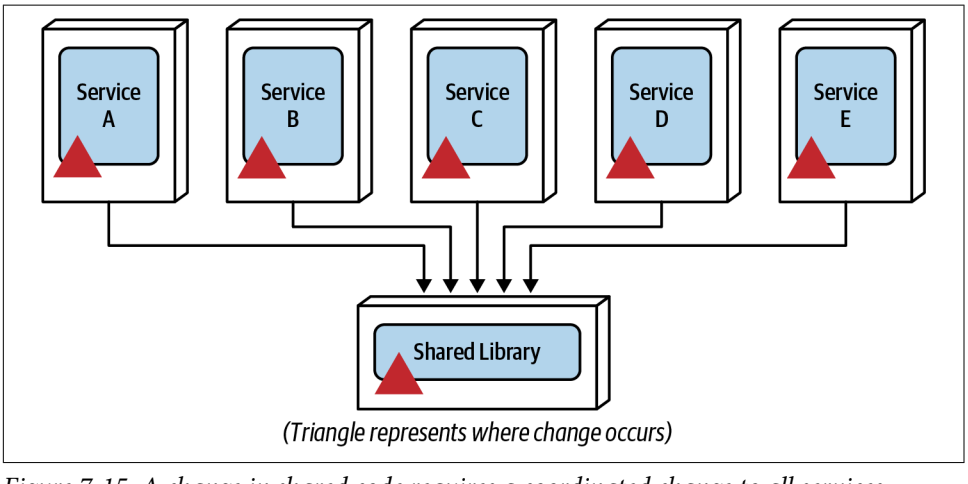


Figure 7-15. A change in shared code requires a coordinated change to all services

Not all uses of shared code drive granularity integration. For example, infrastructure-related cross-cutting functionality such as logging, auditing, authentication, authorization, and monitoring that all services use is *not* a good driver for putting services back together or even moving back to a monolithic architecture. Some of the guidelines for considering shared code as a granularity integrator are as follows:

Specific shared domain functionality
: Shared domain functionality is shared code that contains business logic (as opposed to infrastructure-related cross-cutting functionality). Our recommendation is to consider this factor as a possible granularity integrator if the percentage of shared domain code is relatively high. For example, suppose the common (shared) code for a group of customer-related functionality (profile maintenance, preference maintenance, and adding or removing comments) makes up over 40%

of the collective codebase. Breaking up the collective functionality into separate services would mean that almost half of the source code is in a shared library used only by those three services. In this example it might be wise to consider keeping the collective customer-related functionality in a single consolidated service along with the shared code (particularly if the shared code changes frequently, as discussed next).

Frequent shared code changes
: Regardless of the size of the shared library, frequent changes to shared functionality require frequent coordinated changes to the services using that shared domain functionality. While versioning can sometimes be used to help mitigate coordinated changes, eventually services using that shared functionality will need to adopt the latest version. If the shared code changes frequently, it might be wise to consider consolidating the services using that shared code to help mitigate the complex change coordination of multiple deployment units.

Defects that cannot be versioned
: While versioning can help mitigate coordinated changes and allow for backward compatibility and agility (the ability to respond quickly to change), at times certain business functionality must be applied to all services at the same time (such as a defect or a change in business rules). If this happens frequently, it might be time to consider putting services back together to simplify the changes.

Data Relationships

Another trade-off in the balance between granularity disintegrators and integrators is the relationship between the data that a single consolidated service uses as opposed to the data that separate services would use. This integrator driver assumes that the data resulting from breaking apart a service is not shared, but rather formed into tight bounded contexts within each service to facilitate change control and support overall availability and reliability.

Consider the example in Figure 7-16: a single consolidated service has three functions (A, B, and C) and corresponding data table relationships. The solid lines pointing to the tables represent writes to the tables (hence data ownership), and the dotted lines pointing away from the tables represent read-only access to the table. Performing a mapping operation between the functions and the tables reveals the results shown in Table 7-1, where *owner* implies writes (and corresponding reads) and *access* implies read-only access to a table not owned by that function.

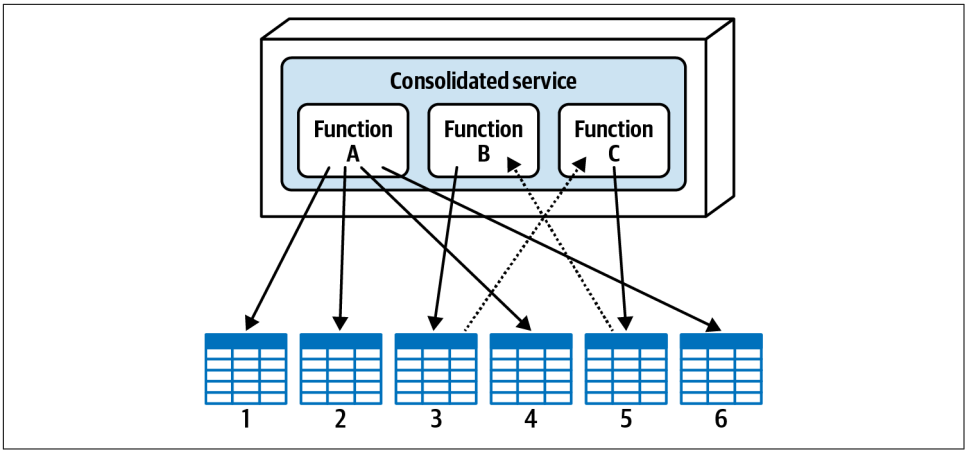


Figure 7-16. The database table relationships of a consolidated service

Table 7-1. Function-to-table mapping

Function	Table 1	Table 2	Table 3	Table 4	Table 5	Table 6
A	owner	owner		owner		owner
B			owner		access	
C			access		owner	

Assume that based on some of the disintegration drivers outlined in the prior section, this service was broken into three separate services (one for each of the functions in the consolidated service); see Figure 7-17. However, breaking apart the single consolidated service into three separate services now requires the corresponding data tables to be associated with each service in a bounded context.

Notice at the top of Figure 7-17 that Service A owns tables 1, 2, 4, and 6 as part of its bounded context; Service B owns table 3; and Service C owns table 5. However, notice in the diagram that every operation in Service B requires access to data in table 5 (owned by Service C), and every operation in Service C requires access to data in table 3 (owned by Service B). Because of the bounded context, Service B cannot simply reach out and directly query table 5, nor can Service C directly query table 3.

To better understand the bounded context and why Service C cannot simply access table 3, say Service B (which owns table 3) decides to make a change to its business rules that requires a column to be removed from table 3. Doing so would break Service C and any other services using table 3. This is why the bounded context concept is so important in highly distributed architectures like microservices. To resolve this issue, Service B would have to *ask* Service C for its data, and Service C would have to ask Service B for its data, resulting in back-and-forth interservice communication between these services, as illustrated at the bottom of Figure 7-17.

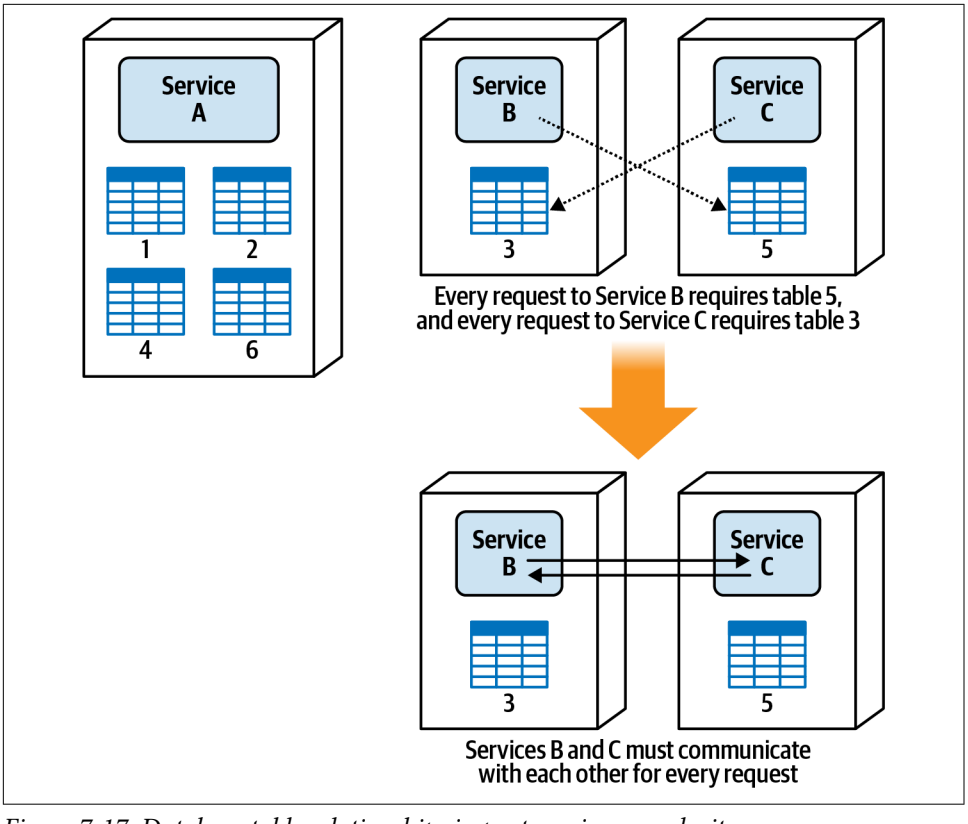


Figure 7-17. Database table relationships impact service granularity

Based on the dependency of the data between services B and C, it would be wise to consolidate those services into a single service to avoid the latency, fault tolerance, and scalability issues associated with the interservice communication between these services, demonstrating that relationships between tables can influence service granularity. We've saved this granularity integration driver for last because it is the one granularity integration driver with the fewest number of trade-offs. While occasionally a migration from a monolithic system requires a refactoring of the way data is organized, in most cases it isn't feasible to reorganize database table entity relationships for the sake of breaking apart a service. We dive into the details about breaking apart data in Chapter 6.

Finding the Right Balance

Finding the right level of service granularity is hard. The secret to getting granularity right is understanding both granularity disintegrators (when to break apart a service) and granularity integrators (when to put them back together), and analyze the corresponding trade-offs between the two. As illustrated in the previous scenarios, this requires an architect to not only identify the trade-offs, but also to collaborate closely with business stakeholders to analyze those trade-offs and arrive at the appropriate solution for service granularity.

Tables 7-2 and 7-3 summarize the drivers for disintegrators and integrators.

Table 7-2. Disintegrator drivers (breaking apart a service)

Disintegrator driver	Reason for applying driver
Service scope	Single-purpose services with tight cohesion
Code volatility	Agility (reduced testing scope and deployment risk)
Scalability	Lower costs and faster responsiveness
Fault tolerance	Better overall uptime
Security access	Better security access control to certain functions
Extensibility	Agility (ease of adding new functionality)

Table 7-3. Integrator drivers (putting services back together)

Integrator driver	Reason for applying driver
Database transactions	Data integrity and consistency
Workflow	Fault tolerance, performance, and reliability
Shared code	Maintainability
Data relationships	Data integrity and correctness

Architects can use the drivers in these tables to form trade-off statements that can then be discussed and resolved by collaborating with a product owner or business sponsor.

Example 1:

> **Architect**: "We want to break apart our service to isolate frequent code changes, but in doing so we won't be able to maintain a database transaction. Which is more important based on our business needs—better *overall agility* (maintainability, testability, and deployability), which translates to faster time-to-market, or stronger *data integrity and consistency*?"
>
> **Project Sponsor**: "Based on our business needs, I'd rather sacrifice a little bit slower time-to-market to have better data integrity and consistency, so let's leave it as a single service for right now."

Example 2:

> **Architect**: "We need to keep the service together to support a database transaction between two operations to ensure data consistency, but that means sensitive functionality in the combined single service will be less secure. Which is more important based on our business needs—better *data consistency* or better *security*?"
>
> **Project Sponsor**: "Our CIO has been through some rough situations with regard to security and protecting sensitive data, and it's on the forefront of their mind and part of almost every discussion. In this case, it's more important to secure sensitive data, so let's keep the services separate and work out how we can mitigate some of the issues with data consistency."

Example 3:

> **Architect**: "We need to break apart our payment service to provide better extensibility for adding new payment methods, but that means we will have increased workflow that will impact the responsiveness when multiple payment types are used for an order (which happens frequently). Which is more important based on our business needs—better extensibility within the payment processing, hence better *agility and overall time-to-market*, or better *responsiveness* for making a payment?"
>
> **Project Sponsor**: "Given that I see us adding only two, maybe three more payment types over the next couple of years, I'd rather have us focus on the overall responsiveness since the customer must wait for payment processing to be complete before the order ID is issued."

Sysops Squad Saga: Ticket Assignment Granularity

`Monday, October 25 11:08`

Once a trouble ticket has been created by a customer and accepted by the system, it must be assigned to a Sysops Squad expert based on their skill set, location, and availability. Ticket assignment involves two main components—a Ticket Assignment component that determines which consultant should be assigned the job, and the Ticket Routing component that locates the Sysops Squad expert, forwards the ticket to the expert's mobile device (via a custom Sysops Squad mobile app), and notifies the expert via an SMS text message that a new ticket has been assigned.

The Sysops Squad development team was having trouble deciding whether these two components (assignment and routing) should be implemented as a single consolidated service or two separate services, as illustrated in Figure 7-18. The development team consulted with Addison (one of the Sysops Squad architects) to help decide which option it should go with.

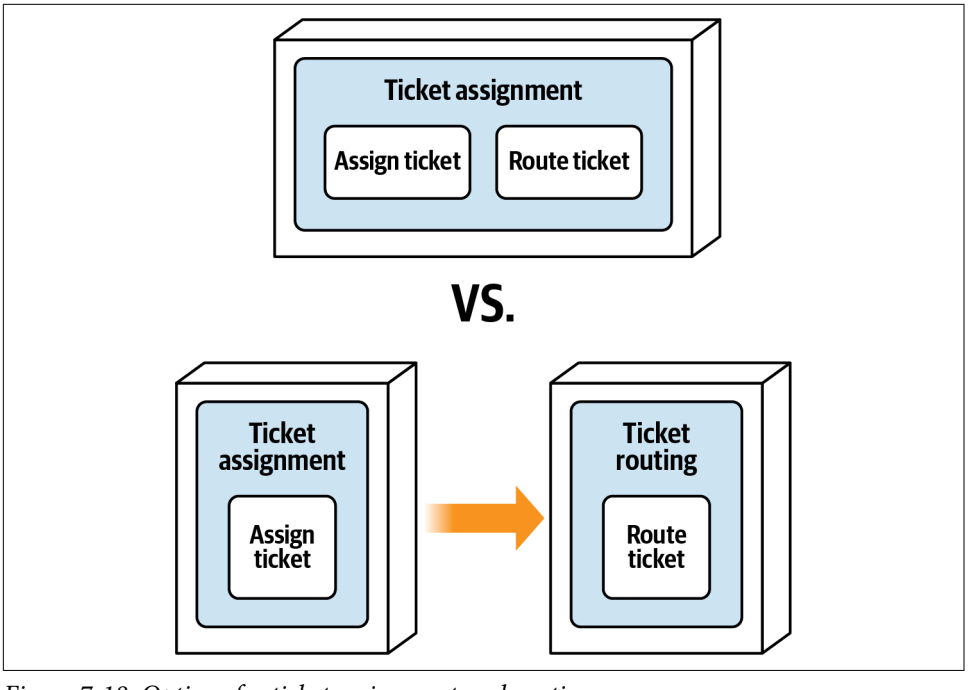


Figure 7-18. Options for ticket assignment and routing

"So you see," said Taylen, "the ticket assignment algorithms are very complex, and therefore should be isolated from the ticket routing functionality. That way, when those algorithms change, I don't have to worry about all of the routing functionality."

"Yes, but how much change is there to those assignment algorithms?" asked Addison. "And how much change do we anticipate in the future?"

"I apply changes to those algorithms at least two to three times a month. I read about volatility-based decomposition, and this situation fits it perfectly," said Taylen.

"But if we separated the assignment and routing functionality into two services, there would need to be constant communication between them," said Skyler. "Furthermore, assignment and routing are really one function, not two."

"No," said Taylen, "they are two separate functions."

"Hold on," said Addison. "I see what Skyler means. Think about it a minute. Once an expert is found that is available within a certain period of time, the ticket is immediately routed to that expert. If no expert is available, the ticket goes back in the queue and waits until an expert can be found."

"Yes, that's right," said Taylen.

"See," said Skyler, "you cannot make a ticket assignment without routing it to the expert. So the two functions are one."

"No, no, no," said Taylen. "You don't understand. If an expert is seen to be available within a certain amount of time, then that expert is assigned. Period. Routing is just a transport thing."

"What happens in the current functionality if a ticket can't be routed to the expert?" asked Addison.

"Then another expert is selected," said Taylen.

"OK, so think about it a minute, Taylen," said Addison. "If assignment and routing are two separate services, then the routing service would have to then communicate back to the assignment service, letting it know that the expert cannot be located and to pick another one. That's a lot of coordination between the two services."

"Yes, but they are still *two* separate functions, not one as Skyler is suggesting," said Taylen.

"I have an idea," said Addison. "Can we all agree that the assignment and routing are two separate activities, but are tightly bound synchronously to each other? Meaning, one function cannot exist without the other?"

"Yes," both Taylen and Skyler replied.

"In that case," said Addison, "let's analyze the trade-offs. Which is more important—isolating the assignment functionality for change control purposes, or combining assignment and routing into a single service for better performance, error handling, and workflow control?"

"Well," said Taylen, "when you put it that way, obviously the single service. But I still want to isolate the assignment code."

"OK," said Addison, "in that case, how about we make three distinct architectural components in the single service. We can delineate assignment, routing, and shared code with separate namespaces in the code. Would that help?"

"Yeah," said Taylen, "that would work. OK, you both win. Let's go with a single service then."

"Taylen," said Addison, "it's not about winning, it's about analyzing the trade-offs to arrive at the most appropriate solution; that's all."

With everyone agreeing to a single service for assignment and routing, Addison wrote the following architecture decision record (ADR) for this decision:

> *ADR: Consolidated Service for Ticket Assignment and Routing*
>
> *Context*
> Once a ticket is created and accepted by the system, it must be assigned to an expert and then routed to that expert's mobile device. This can be done through a single consolidated ticket assignment service or separate services for ticket assignment and ticket routing.

Decision

We will create a single consolidated ticket assignment service for the assignment and routing functions of the ticket.

Tickets are immediately routed to the Sysops Squad expert once they are assigned, so these two operations are tightly bound and dependent each other.

Both functions must scale the same, so there are no throughput differences between these services, nor is back-pressure needed between these functions.

Since both functions are fully dependent on each other, fault tolerance is not a driver for breaking these functions apart.

Making these functions separate services would require workflow between them, resulting in performance, fault tolerance, and possible reliability issues.

Consequences

Changes to the assignment algorithm (which occur on a regular basis) and changes to the routing mechanism (infrequent change) would require testing and deployment of both functions, resulting in increased testing scope and deployment risk.

Sysops Squad Saga: Customer Registration Granularity

`Friday January 14, 13:15`

Customers must register with the system to gain access to the Sysops Squad support plan. During registration, customers must provide profile information (name, address, business name if applicable, and so on), credit card information (which is billed on a monthly basis), password and security question information, and a list of products purchased they would like to have covered under the Sysops Squad support plan.

Some members of the development team insisted that this should be a single consolidated Customer Service containing all of the customer information, yet other members of the team disagreed and thought that there should be a separate service for each of these functions (a Profile service, Credit Card service, Password service, and a Supported Product service). Skyler, having prior experience in PCI and PII data, thought that the credit card and password information should be a separate service from the rest, and hence only two services (a Profile service containing profile and product information and a separate Customer Secure service containing credit card and password information). These three options are illustrated in Figure 7-19.

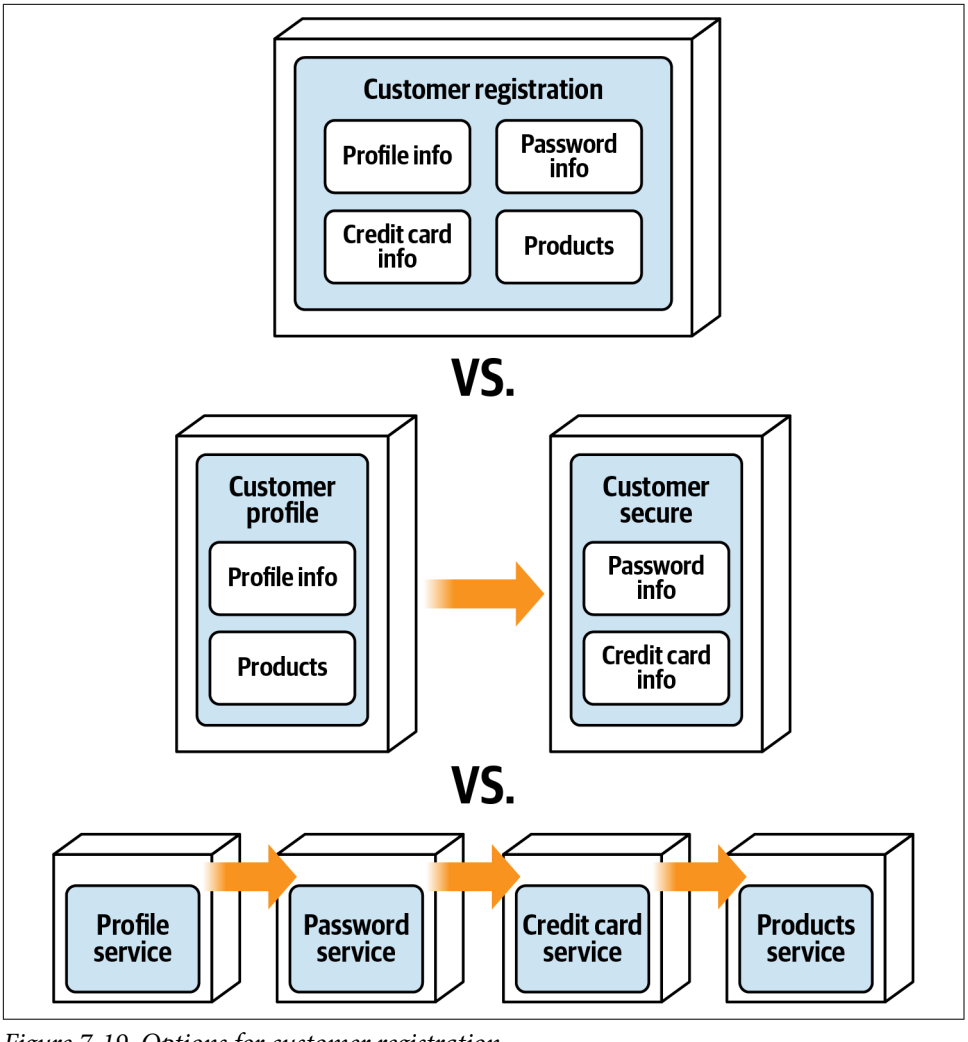


Figure 7-19. Options for customer registration

Because Addison was busy with the core ticketing functionality, the development team asked for Austen's help in resolving this granularity issue. Anticipating this will not be an easy decision, particularly since it involved security, Austen scheduled a meeting with Parker, (the product owner), and Sam, the Penultimate Electronics security expert to discuss these options.

"OK, so what can we do for you?" asked Parker.

"Well," said Austen, "we are struggling with how many services to create for registering customers and maintaining customer-related information, You see, there are four main pieces of data we are dealing with here: profile info, credit card info, password info, and purchased product info."

"Whoa, hold on now," interrupted Sam. "You know that credit card and password information *must* be secure, right?"

"Of course we know it has to be secure," said Austen. "What we're struggling with is the fact that there's a single customer registration API to the backend, so if we have separate services they all have to be coordinated together when registering a customer, which would require a distributed transaction."

"What do you mean by that?" asked Parker.

"Well," said Austen, "we wouldn't be able to synchronize all of the data together as one atomic unit of work."

"That's not an option," said Parker. "All of the customer information is either saved in the database, or it's not. Let me put it another way. We absolutely *cannot* have the situation where we have a customer record without a corresponding credit card or password record. *Ever*."

"OK, but what about securing the credit card and password information?" asked Sam. "Seems to me, having separate services would allow much better security control access to that type of sensitive information."

"I think I may have an idea." said Austen. "The credit card information is tokenized in the database, right?"

"Tokenized *and* encrypted," said Sam.

"Great. And the password information?" asked Austen.

"The same," said Sam.

"OK," said Austen, "so it seems to me that what we really need to focus on here is controlling access to the password and credit card information separate from the other customer-related requests—you know, like getting and updating profile information, and so on."

"I think I see where you are coming from with your problem," said Parker. "You're telling me that if you separate all of this functionality into separate services, you can better secure access to sensitive data, but you cannot guarantee my all-or-nothing requirement. Am I right?"

"Exactly. That's the trade-off," said Austen.

"Hold on," said Sam. "Are you using the Tortoise security libraries to secure the API calls?"

"Yes. We use those libraries not only at the API layer, but also within each service to control access through the service mesh. So essentially it's a double-check," said Austen.

"Hmmm," said Sam. "OK, I'm good with a single service providing you use the Tortoise security framework."

"Me too, providing we can still have the all-or-nothing customer registration process," said Parker.

"Then I think we are all in agreement that the all-or-nothing customer registration is an absolute requirement and we will maintain multilevel security access using Tortoise," said Austen.

"Agreed," said Parker.

"Agreed," said Sam.

Parker noticed how Austen handled the meeting by facilitating the conversation rather than controlling it. This was an important lesson as an architect in identifying, understanding, and negotiating trade-offs. Parker also better understood the difference between design versus architecture in that security can be controlled through *design* (use of a custom library with special encryption) rather than *architecture* (breaking up functionality into separate deployment units).

Based on the conversation with Parker and Sam, Austen made the decision that customer-related functionality would be managed through a single consolidated domain service (rather than separately deployed services) and wrote the following ADR for this decision:

> *ADR: Consolidated Service for Customer-Related Functionality*
>
> *Context*
> Customers must register with the system to gain access to the Sysops Squad support plan. During registration, customers must provide profile information, credit card information, password information, and products purchased. This can be done through a single consolidated customer service, a separate service for each of these functions, or a separate service for sensitive and nonsensitive data.
>
> *Decision*
> We will create a single consolidated customer service for profile, credit card, password, and products supported.
>
> Customer registration and unsubscribe functionality *requires* a single atomic unit of work. A single service would support ACID transactions to meet this requirement, whereas separate services would not.
>
> Use of the Tortoise security libraries in the API layer and the service mesh will mitigate security access risk to sensitive information.
>
> *Consequences*
> We will require the Tortoise security library to ensure security access in both the API gateway and the service mesh.
>
> Because it's a single service, changes to source code for profile info, credit card, password, or products purchased will increase testing scope and increase deployment risk.
>
> The combined functionality (profile, credit card, password, and products purchased) will have to scale as one unit.
>
> The trade-off discussed in a meeting with the product owner and security expert is *transactionality* versus *security*. Breaking the customer functionality into separate services

provides better security access, but doesn't support the "all-or-nothing" database transaction required for customer registration or unsubscribing. However, the security concerns are mitigated through the use the custom Tortoise security library.

About the Authors

Neal Ford is a director, software architect, and meme wrangler at Thoughtworks, a software company and a community of passionate, purpose-led individuals who think disruptively to deliver technology that addresses the toughest challenges, all while seeking to revolutionize the IT industry and create positive social change. He's an internationally recognized expert on software development and delivery, especially in the intersection of Agile engineering techniques and software architecture. Neal has authored seven books (and counting), a number of magazine articles, and dozens of video presentations and spoken at hundreds of developers conferences worldwide. His topics include software architecture, continuous delivery, functional programming, cutting-edge software innovations, and a business-focused book and video on improving technical presentations. Check out his website, *Nealford.com*.

Mark Richards is an experienced, hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service-oriented architectures, and distributed systems in a variety of technologies. He has been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. Mark is the author of numerous technical books and videos, including the *Fundamentals of Software Architecture*, the "Software Architecture Fundamentals" video series, and several books and videos on microservices as well as enterprise messaging. Mark is also a conference speaker and trainer and has spoken at hundreds of conferences and user groups around the world on a variety of enterprise-related technical topics.

Pramod Sadalage is director of data and DevOps at Thoughtworks. His expertise includes application development, Agile database development, evolutionary database design, algorithm design, and database administration.

Zhamak Dehghani is director of emerging technologies at Thoughtworks. Previously, she worked at Silverbrook Research as a principal software engineer, and Fox Technology as a senior software engineer.

Colophon

The animal on the cover of *Software Architecture: The Hard Parts* is a black-rumped golden flameback woodpecker (*Dinopium benghalense*), a striking species of woodpecker found throughout the plains, foothills, forests, and urban areas of the Indian subcontinent.

This bird's golden back is set atop a black shoulder and tail, the reason for its pyro-inspired name. Adults have red crowns with black-and-white spotted heads and breasts, with a black stripe running from their eyes to the back of their heads. Like other common, small-billed woodpeckers, the black-rumped golden flameback has a straight pointed bill, a stiff tail to provide support against tree trunks, and four-toed feet—two toes pointing forward and two backward. As if its markings weren't distinctive enough, the black-rumped golden flameback woodpecker is often detected by its call of "ki-ki-ki-ki-ki," which steadily increases in pace.

This woodpecker feeds on insects, such as red ant and beetle larvae, underneath tree bark using its pointed bill and long tongue. They have been observed visiting termite mounds and even feeding on the nectar of flowers. The golden flameback also adapts well to urban habitats, subsisting on readily available fallen fruit and food scraps.

Considered relatively common in India, this bird's current conservation status is listed as being of "least concern." Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover image is a color illustration by Karen Montgomery, based on a black and white engraving from *Shaw's Zoology*. The cover fonts are URW Typewriter and Guardian Sans. The text fonts are Adobe Minion Pro and Myriad Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.