Chapter 22. Analyzing Architecture Risk

Every architecture comes with risks: some operational (such as availability, scalability, and data integrity), some structural (such as static coupling between logical components). Analyzing architecture risk is one of architects' most important activities, allowing them to address deficiencies and structural decay within the architecture and take corrective action. In this chapter, we show you some key techniques and practices for quantifying, assessing, and identifying risk, and introduce an activity called *risk storming*.

Risk Matrix

The first thing to determine in assessing architecture risk is its level: whether risk to a particular part of the architecture is low, medium, or high. The challenge here is that assessing risk can be *subjective*. One architect might hold the *opinion* that some aspect of the architecture is high risk, while another architect's *opinion* might be that the same aspect is medium risk. We italicize *opinion* here to emphasize the subjectivity of assessing risk. Fortunately, architects have a useful risk-assessment matrix that helps us make risk more measurable.

The architecture risk-assessment matrix (Figure 22-1) uses two dimensions to qualify risk: the overall impact of the risk involved and the likelihood of that risk occurring. The architect rates each dimension as low (1), medium (2), or high (3), then multiplies the numbers within each intersection of the matrix. This provides a numerical representation of risk, making the process of qualifying risk more *objective*. Ratings of 1 and 2 are considered low risk (usually depicted in green), numbers 3 and 4 are considered medium risk (usually yellow), and numbers 6 through 9 are considered high risk (usually red). Using shading can be helpful for grayscale rendering and for people unable to distinguish colors.

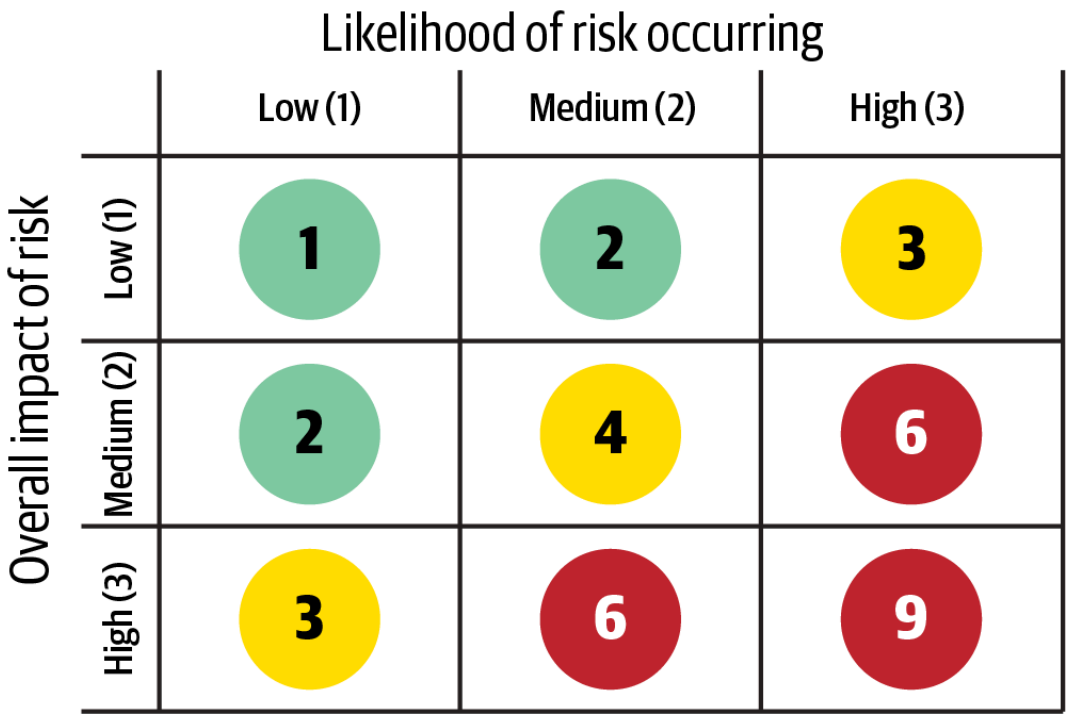

Figure 22-1. Matrix for determining architecture risk

To show you its utility, we'll use the risk matrix in an example. Suppose you're concerned about the availability of your application's primary central database.

Tip

When using this matrix to qualify risk, consider impact first and likelihood second. If you are unsure of the likelihood, use a high (3) rating until you can confirm.

First, you'll consider overall impact: what happens if the database goes down or becomes unavailable? Let's say you might deem the impact high risk and map that risk to the last row of the matrix in Figure 22-1 as a 3 (medium), 6 (high), or 9, (high). However, when you consider the second dimension—the likelihood of that risk occurring—you realize that the database is on highly available servers in a clustered configuration, and the *likelihood* that the database would become unavailable is low (first column in the matrix). The intersection between high impact and low likelihood gives you an overall risk rating of 3 (medium risk) for availability in the primary central database.

Risk Assessments

You can use the risk matrix to build what is called a *risk assessment*: a summarized report of an architecture's overall risk, with meaningful assessment criteria based on a context (services, subdomain areas, or domain areas of a system). We've performed many risk assessments, and have found architectural characteristics to be excellent risk-assessment criteria. Why spend time analyzing performance risk when the system's critical architectural characteristics are scalability, elasticity, and data integrity? Knowing characteristics such as those described in Chapter 4 is the first step in analyzing architectural risk.

Tip

The architectural characteristics that are most critical for the architecture to support make great risk-assessment criteria.

The basic format of a risk-assessment report is shown in Figure 22-2. Here, 1 and 2 represent low risk, 3 and 4 represent medium risk, and 6 and 9 represent high risk. The risk criteria run along the left side of the spreadsheet, and the context runs across the top.

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6	1	2	11
Availability	3	4	2	1	10
Performance	4	2	3	6	15
Security	6	3	1	1	11
Data integrity	9	6	1	1	17
TOTAL RISK	24	21	8	11	

Figure 22-2. Example of a standard risk assessment

We use an ecommerce ordering system as the example for this risk assessment. It assesses five criteria, representing the system's critical architectural characteristics. Across the top are four different contexts, each representing a separate domain (customer registration, catalog checkout, order fulfillment, and order shipping). A domain or subdomain context works well; analyzing risk at the level of services is usually too fine-grained and doesn't account for risk involving communication or coordination between multiple services.

The nice thing about using quantified risk is that it considers both the risk criteria and the context. For example, in Figure 22-2, the total accumulated risk for data integrity is 17, making it the highest risk area from a criteria standpoint. The accumulated risk for availability is only 10 (the lowest risk criteria-wise). In terms of the relative risk of each context area, however, customer registration is the domain that carries the highest risk context-wise, whereas order fulfillment is the lowest-risk context. This is useful information when determining priorities and where to put additional effort into reducing risk.

This risk assessment example contains all of the risk-analysis results, but it's sometimes useful to filter out details to highlight certain problems. For example, suppose that you, as the architect of this system, are in a meeting, presenting to stakeholders about the high-risk areas of the system. Rather than presenting the full risk assessment, as Figure 22-2 does, you could filter out the low- and medium-risk areas (the noise) to highlight the high-risk areas (the signal). Improving the overall signal-to-noise ratio lets you deliver a more effective, less distracting message. Figure 22-3 shows a filtered version of the same risk assessment. Compare the two images to see how much clearer the message is with the second filtered assessment.

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability		6			6
Availability					0
Performance				6	6
Security	6				6
Data integrity	9	6			15
TOTAL RISK	15	12	0	6	

Figure 22-3. Filtering the risk assessment to show only high risks

One problem with the full version of the risk assessment is that it only shows a snapshot in time, not whether things are improving or getting worse. In other words, Figure 22-2 does not show the *direction of risk*. You can determine the direction of risk by using continuous measurements through fitness functions, as described in Chapter 6. Objectively analyzing each risk criterion lets you observe trends to spot the direction of each risk criterion.

In Figure 22-4, we add a third dimension to the risk assessment: *direction*. We use a right-side-up triangle to indicate that the risk for that particular criteria and context is getting worse—the tip of the triangle points up, toward a *higher* number. Conversely, we use an upside-down triangle to indicate that the risk is lessening (in other words, the tip is pointing down to a *lower* number). Finally, we use a circle to represent that the risk is not moving—getting neither better nor worse. This can get confusing, so we always recommend including a key when using any sort of symbol to represent direction.

RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6	1	2	11
Availability	3	4	2	1	10
Performance	4	2	3	6	15
Security	6	3	1	1	11
Data integrity	9	6	1	1	17
TOTAL RISK	24	21	8	11	

Figure 22-4. Showing direction of risk using triangles

This revised architectural risk assessment, now showing direction, tells a different story than the original. First, we can see that data integrity is getting worse based on continuous measurements (triangle pointing up) for catalog checkout, order fulfillment, and order shipping, which could indicate a database issue. However,

security and availability are getting better overall (triangle pointing down) for customer registration and catalog checkout, indicating improvements in those areas.

Next, we describe a process called *risk storming* that teams can use to determine *how* to identify the risk level for particular contexts and criteria.

Risk Storming

No architect can singlehandedly determine the overall risk of a system, for two reasons. First, an architect working alone might miss or overlook a risk area; second, very few architects have full knowledge of *every* part of the system. This is where risk storming can help.

Risk storming is a collaborative exercise to determine architectural risk within a specific dimension (either context or criteria). While most risk-storming efforts involve multiple architects, we strongly recommend including senior developers and tech leads as well. Not only will they provide an implementation perspective on architectural risk, but involving them helps them better understand the architecture.

Risk storming involves three phases: identification, consensus, and mitigation. In the individual phase (phase 1), all participants, working alone, use the risk matrix to assign risk to various areas of the architecture. This individual phase of risk storming is essential so that participants don't influence other participants or direct people's attention away from particular areas of the architecture. In the two collaborative phases, all participants work together to gain consensus on what the risk areas are and discuss them (phase 2), and come up with solutions for mitigating the risk (phase 3).

All three phases utilize a comprehensive or contextual architecture diagram (see Chapter 23). The architect conducting the risk-storming effort—we'll call this person the *facilitator*—is responsible for sending all participants updated diagrams for the risk-storming session.

Figure 22-5 shows an example architecture we'll use to illustrate the risk-storming process. In this architecture, an Elastic Load Balancer forwards a request to each EC2 instance containing the web servers (Nginx) and application services. The application services make calls to a MySQL database, a Redis cache, and a MongoDB database (for logging). They also make calls to the Push Expansion Servers, which in turn all interface with the MySQL database, Redis cache, and MongoDB logging facility. (Don't worry if you don't understand all of these products and buzzwords—this overly vague, generalized architecture is only meant to illustrate how risk storming works.)

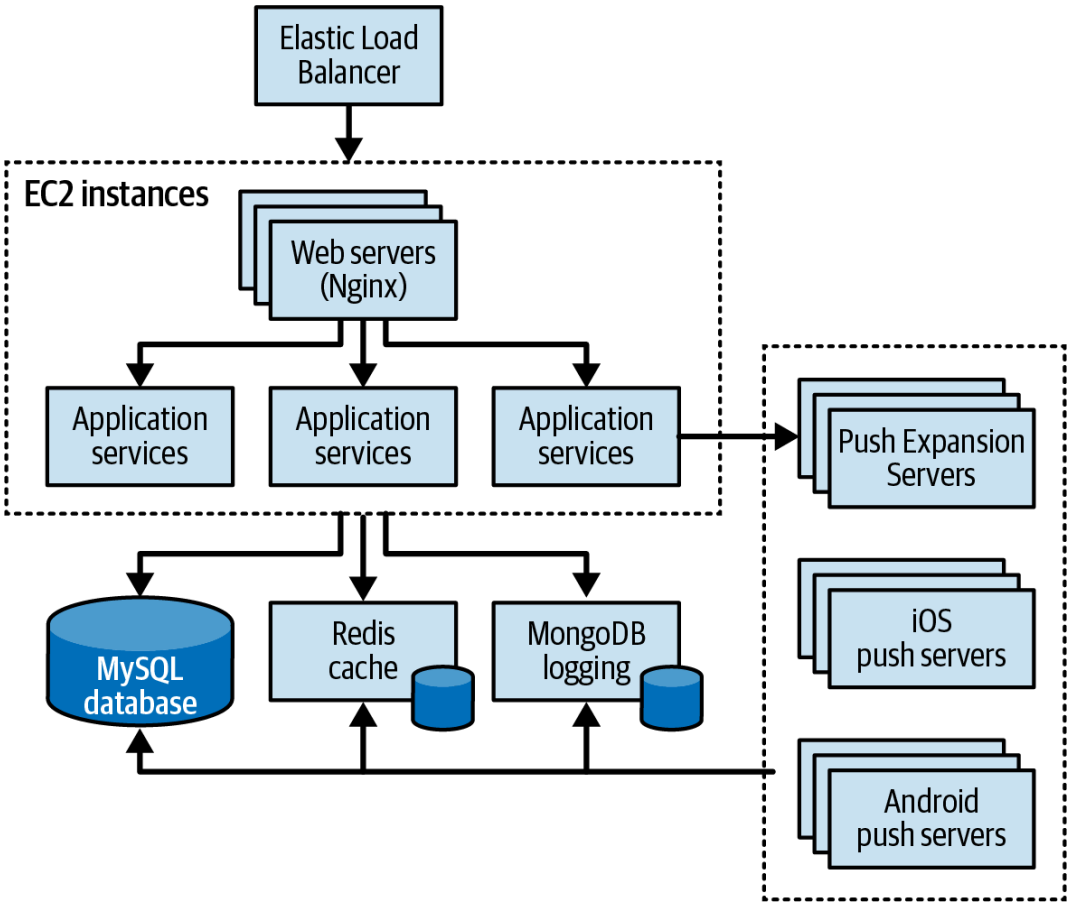

Figure 22-5. Example architecture diagram for risk storming session

We'll begin with phase 1.

Phase 1: Identification

The *identification* phase of risk storming involves each participant individually identifying areas of risk within the architecture. It's critical that, in this phase, each participant should record their unbiased view of the risk involved, without being redirected or swayed by other participants. The identification phase has three steps:

1. The facilitator sends all participants an invitation to the collaborative phases. The invitation contains the architecture diagram (or where to find it), the risk criteria and context to be analyzed, and the date, time, and location (physical or virtual) of the collaborative session, along with any other logistical details.

2. Participants use the risk matrix to analyze the architecture risks individually.

3. Participants classify each risk as low (1–2), medium (3–4), or high (6–9), and write the numbers on small green, yellow, or red sticky notes.

Most risk-storming efforts analyze only one particular criterion or context (such as "Where are our security risks?" or "What areas are at risk within customer registration?"). However, if there are issues with staff availability or timing, the risk-assessment team may have to analyze multiple dimensions within a specific context (such as performance and scalability). In these cases, participants typically write the specific criterion next to the risk number on the sticky notes. For example, suppose three participants identify risk with a central database. All three participants identify the risk as high (6), but one participant sees this as a risk to

availability, while the other two see it as a risk to performance. The participants should discuss these two criteria separately.

Tip

Whenever possible, restrict risk-storming efforts to a single criterion or context. This allows participants to focus their attention on that specific dimension and avoids confusion about what the actual risk is.

Phase 2: Consensus

The *consensus* phase of risk storming is highly collaborative, with the goal of gaining consensus among all participants regarding the risk or risks within the architecture. This activity is most effective when the facilitator posts a large printed architecture diagram on the wall (or an electronic version on a large screen). When the participants arrive at the risk-storming session, the facilitator instructs them to begin placing their risk-level sticky notes on the relevant area of the architecture diagram (see Figure 22-6).

Once all of the sticky notes are in place, the collaborative phase can begin. The goal here is to analyze the risk areas as a team and reach a consensus about their risk level. In the example pictured in Figure 22-6, the team has identified several areas of risk. (The actual criteria aren't important for this example.) We can see that:

- Two participants identified the Elastic Load Balancer as medium risk (3), whereas one participant identified it as high risk (6).
- One participant identified the Push Expansion Servers as high risk (9).
- Three participants identified the MySQL database as medium risk (3).
- One participant identified the Redis cache as high risk (9).
- Three participants identified MongoDB logging as low risk (2).
- No one identified any other areas of the architecture as carrying any risk, so there are no sticky notes on any other areas.

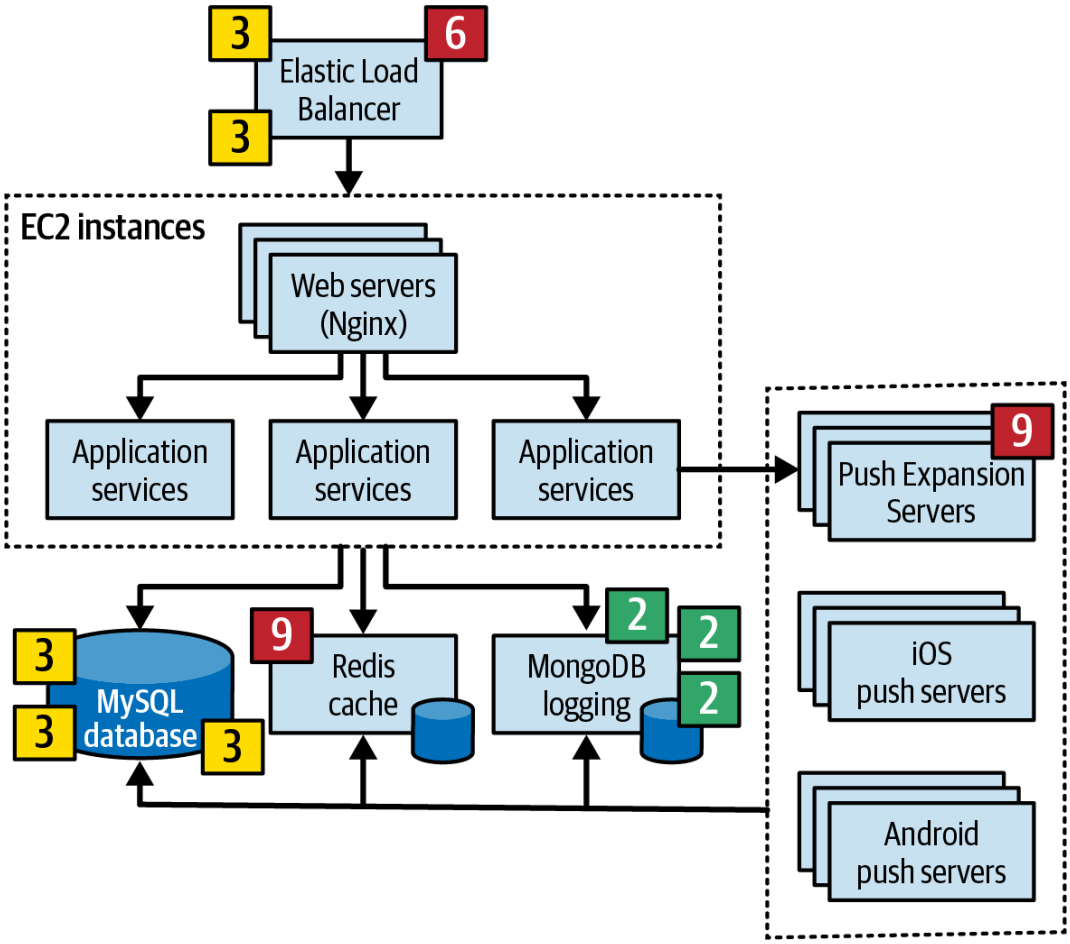

Figure 22-6. Initial identification of risk areas

The MySQL database and MongoDB logging need no further discussion in this session, since all participants agree on their risk levels. However, there's a difference of opinion about the Elastic Load Balancer, and the Push Expansion Servers and Redis cache were only identified as risks by one participant each. Working through these discrepancies is what the collaborative phase is all about.

Two participants (Austen and Logan) identified the Elastic Load Balancer as medium risk (3); one (Addison) identified it as high risk (6). Austen and Logan ask Addison why they identified the risk as high. Addison responds that if the Elastic Load Balancer goes down, the entire system will become inaccessible. While this is true—and brings the *impact* rating up to high—the other two participants convince Addison that there is low risk of this happening due to clustering. Addison agrees, and the group brings the *likelihood* risk level down to a medium (3).

This could have gone a different way, however. If Austen and Logan missed a particular aspect of risk in the Elastic Load Balancer that Addison saw, Addison might have convinced the other two participants to classify this risk's level as high instead of medium. That's why the collaboration phase of risk storming is so important.

One participant identified the Push Expansion Servers as high risk (9), but no other participant identified any risk in this area of the architecture at all. The person who identified the risk explains that they rated the risk as high because they've had bad experiences with Push Expansion Servers continually crashing under high loads similar to this architecture's load. This example shows the value of risk storming—without that participant's involvement, no one would have seen the high risk until well into production.

The Redis cache is an interesting case. One participant, a developer named Devon, identified it as high risk (9), but no one else saw that cache as having any risk. When the other participants ask Devon about their rationale for rating this risk as high, Devon responds, “What’s a Redis cache?” Whenever a risk-storming participant identifies a technology as unknown to them, that area automatically gets assigned a high risk level (9).

Tip

Always assign unproven or unknown technologies the highest risk rating (9), since the risk matrix cannot be used for this criterion or context.

The example of the Redis cache illustrates why it’s important to bring developers into risk-storming sessions. The fact that this participant didn’t know a given technology is valuable information for the architect about overall risk. The architect might decide to change the technology or incur training costs to bring the development team up to speed.

This phase continues until all participants agree on the risk areas identified. Once all the sticky notes are consolidated, this phase ends. Its final outcome is shown in Figure 22-7.

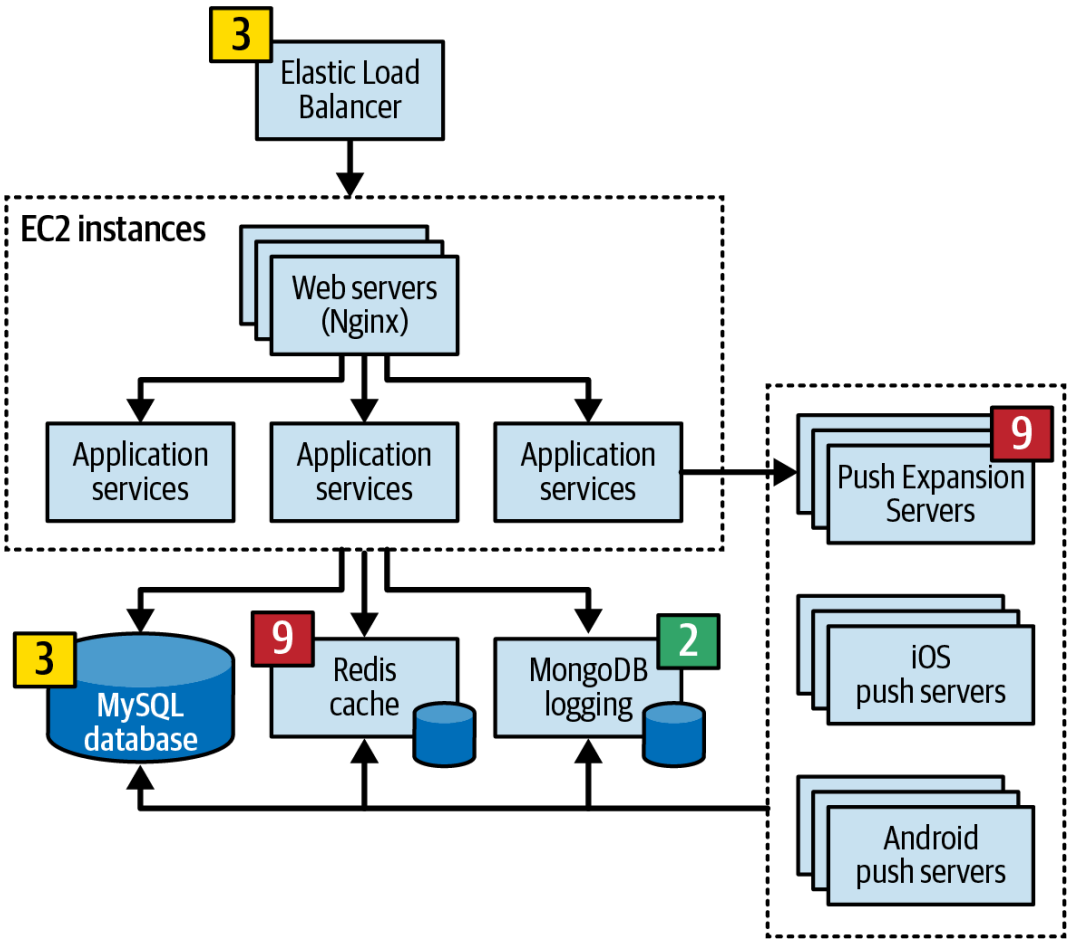

Figure 22-7. Consensus of risk areas

Phase 3: Risk Mitigation

Once all participants agree on the risk levels of the architecture, the *risk mitigation* phase begins. Mitigating risk usually involves changing certain areas of the architecture that otherwise might have been deemed perfect the way they were.

This phase, which is also collaborative, seeks ways to reduce or eliminate the risks identified in the second phase. Depending on the risks identified, the original architecture may need to be completely changed, or the changes could be limited to straightforward architectural refactoring in specific areas, such as adding a queue for backpressure to reduce a throughput bottleneck.

Whatever the changes required, the risk mitigation phase usually incurs additional costs. For that reason, it's important for this phase to involve key business stakeholders with the authority to decide whether the cost of a given mitigation solution outweighs the risk.

For example, suppose that, in our example risk-storming session the team identifies the central database as having medium risk (4) with regard to overall system availability. The participants agree that clustering the database and breaking it into separate physical databases would mitigate that risk. However, that solution would cost $50,000. The facilitating architect meets with key business stakeholders, including the owner, to discuss the trade-offs of availability risk and cost. The business owner decides that the price tag is too high and that the cost does not outweigh the risk of availability. The architect then suggests a different approach: instead of expensive clustering, what about splitting the database into two separate domain-based databases? This solution would only cost $16,000, while still reducing the availability risk. The stakeholders agree to this compromise.

This scenario shows how risk storming shapes not only the overall architecture, but also the negotiations between architects and business stakeholders. In combination with the risk assessments we described at the start of this chapter, risk storming is an excellent vehicle for identifying and tracking risk, improving the architecture, and structuring negotiations between key stakeholders.

User-Story Risk Analysis

Risk storming is useful in many aspects of software development aside from identifying architectural risk. For example, a development team could use risk storming to determine the overall risk associated with user-story completion within a given iteration (and consequently the overall risk assessment of that iteration) during story grooming. Using the same risk matrix, the team can identify user-story risk by identifying the overall impact if the story is not completed within the iteration, and the likelihood that the story will not be completed in the current iteration. Then they can identify high-risk stories, track them carefully, and better prioritize them.

Risk-Storming Use Case

To illustrate the power of risk storming and how it can improve overall architecture, let's consider the example of a support system for a call center where nurses advise patients on various health conditions. The system requirements are:

- A third-party diagnostics engine will serve up questions and guide nurses and patients through their medical issues. This engine can handle about 500 requests a second.
- Patients can either call in using the call center to speak to a nurse or use a self-service website that accesses the same diagnostic engine directly.

- The system must support 250 concurrent nurses nationwide and up to hundreds of thousands of concurrent self-service patients nationwide.
- Nurses can access patients' medical records through a medical records exchange, but patients cannot access their own medical records.
- The system must be HIPAA (Health Insurance Portability and Accountability Act) compliant, meaning it's essential that no one but nurses can access patients' medical records. The self-service option cannot guarantee HIPAA compliance.
- The system needs to be able to handle high volume during cold, flu, and COVID outbreaks.
- Calls are routed to nurses based on each nurse's skill profile (such as languages spoken or medical specializations).

After analyzing these requirements, Logan, the architect responsible for this system, creates the high-level architecture diagrammed in Figure 22-8. This architecture has three separate web-based UIs: one for self-service, one for nurses receiving calls, and one for administrative staff to add and maintain nurse profiles and configuration settings. The call-center portion of the system consists of a `Call Accepter` service, which receives calls, and the `Call Router` service, which routes the caller to the next available nurse based on the nurse's skill profile. The `Call Router` service accesses the central database to get nurse profile information. Central to this architecture is a diagnostics-system API Gateway, which performs security checks and directs requests to the appropriate backend service.

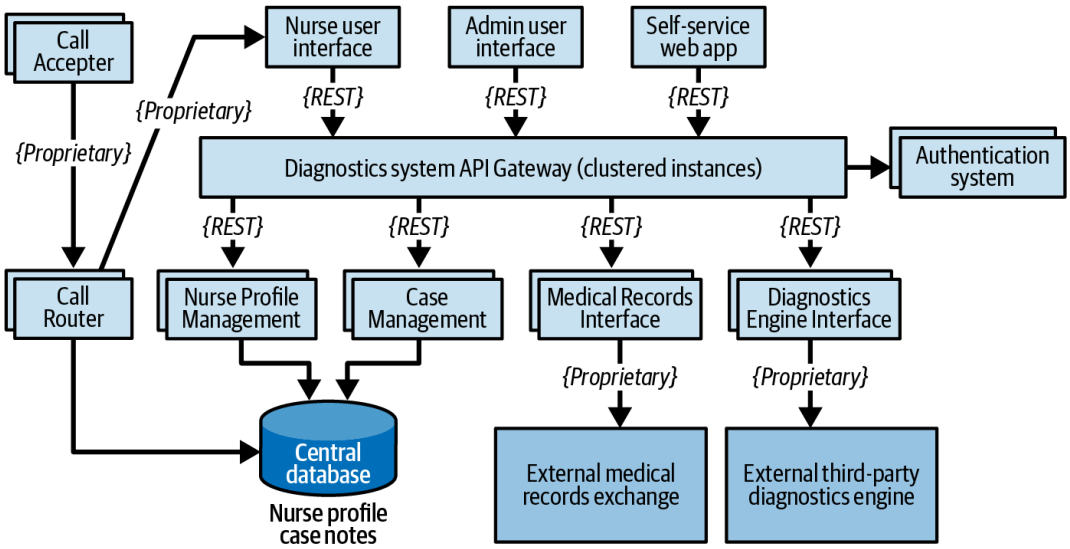

Figure 22-8. High-level architecture for a nursing-hotline diagnostics system

The four main services in this system are the `Case Management` service, the `Nurse Profile Management` service, the `Medical Records Interface` service to the medical records exchange, and the external third-party `Diagnostics Engine Interface` service. All communications use REST, except for proprietary protocols to the external systems and call-center services. The areas the architecture must support are, in summary, availability, elasticity, and security.

After many reviews, Logan believes the architecture is ready for implementation. However, being a responsible and effective architect, Logan decides to hold a risk-storming exercise.

Availability

As facilitator, Logan decides to focus the first risk-storming exercise on availability, which is critical to the system's success. After the identification and collaboration phases, the participants come up with the following risk areas (illustrated in Figure 22-9):

- Central database availability: high risk (6) due to high impact (3) and medium likelihood (2) of the database being unavailable when needed.
- `Diagnostics Engine` availability: high risk (9) due to high impact (3) and unknown likelihood (3) of unavailability.
- `Medical Records Interface` availability: low risk (2); this component is not required to determine a particular medical outcome.
- The team has deemed no other parts of the system to be availability risks, since the architecture includes multiple instances of each service and clusters the API Gateway.

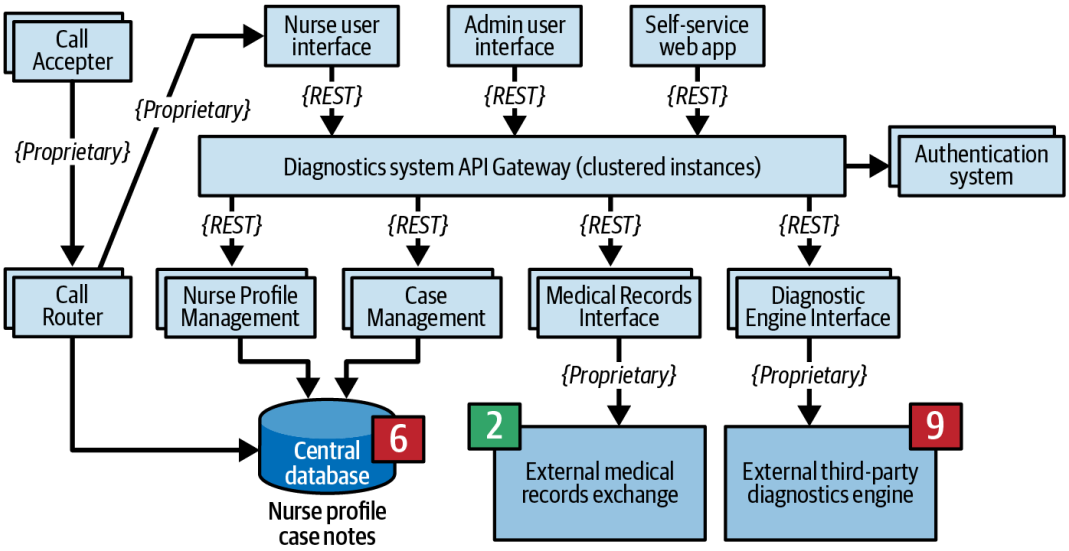

Figure 22-9. Availability risk areas, as identified by the risk-storming team

All participants agree that if the database went down, nurses could write down case notes manually, but the call router could not function. To mitigate this risk, they collectively decide to break apart the single physical database into two separate databases: one clustered database containing nurse-profile information, and one single-instance database for case notes. Not only does this architecture change address the database availability concerns, it also helps secure the case notes.

It's much harder to mitigate availability risk in external systems (in this case, the `Diagnostics Engine` and `Medical Records Interface`) because they are controlled by a third party. The team decides to research whether these systems have published service-level agreements (SLAs) or service-level objectives (SLOs). An SLA is usually a legally binding contractual agreement; an SLO is usually not legally binding. They find SLAs for both systems. The `Diagnostics Engine` SLA guarantees 99.99% availability (that's 52.60 minutes of downtime per year), and the `Medical Records Interface` guarantees at 99.90% availability (that's 8.77 hours of downtime per year). Based on this analysis, this information was enough for the risk assessment team to remove the identified risk.

After this risk-storming session, the team changes the architecture, as illustrated in Figure 22-10, creating two databases and adding the SLAs to the architecture diagram.

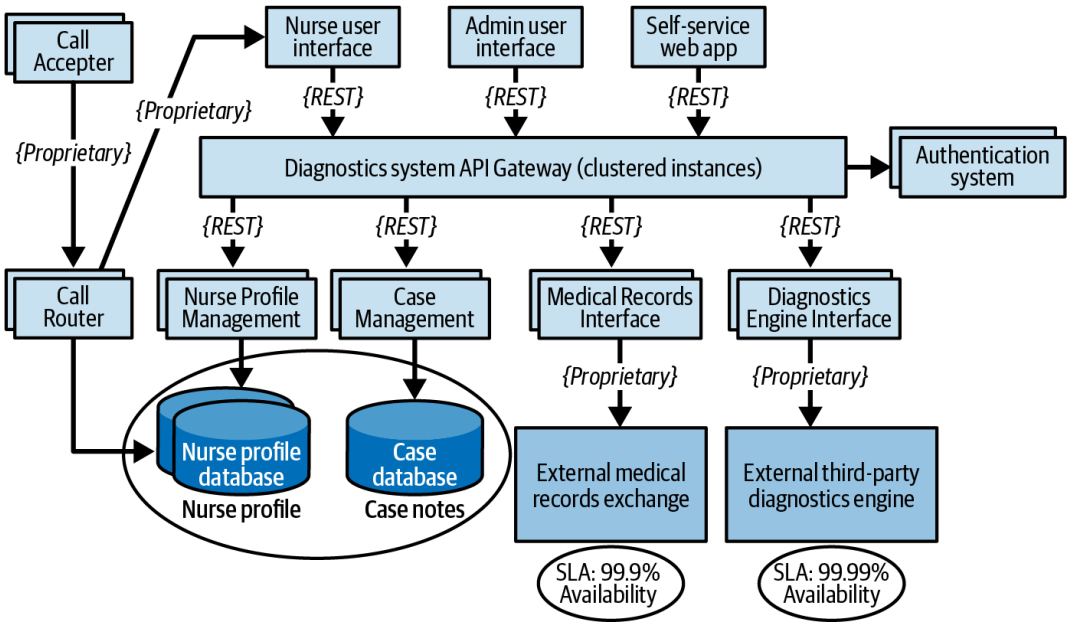

Figure 22-10. Availability risk areas can be mitigated by using separate databases

Elasticity

The second risk-storming exercise focuses on elasticity—spikes in user load (otherwise known as variable scalability). Although there are only 250 nurses (which means no more than 250 nurses will be accessing the `Diagnostics Engine`), the self-service portion of the system can access the `Diagnostics Engine` as well, which significantly increases the number of requests to the `Diagnostics Engine` interface. The risk-storming participants are concerned about flu season and COVID outbreaks, which will significantly increase the anticipated load on the system.

The participants unanimously identify the `Diagnostics Engine` interface as high risk (9). Since it can handle only 500 requests per second, they correctly calculate a high risk that it won't be able to keep up with the anticipated throughput, particularly with REST as its interface protocol.

The team decided that one way to mitigate this risk is to use asynchronous queues (messaging) for communication between the API Gateway and the `Diagnostics Engine` interface. This will provide a backpressure point if calls to the `Diagnostics Engine` get backed up. While this is a good practice, it still doesn't quite mitigate all the risk: nurses and self-service patients will still be waiting too long for responses from the `Diagnostics Engine`, and their requests will likely time out.

The participants decide to use a pattern known as the *Ambulance* pattern to separate these requests by using two message channels instead of just one. This would let the system prioritize nurses' requests over self-service requests. This would help mitigate the risk, but still doesn't address wait times. After more discussion, the group decides to reduce outbreak-related calls to the `Diagnostics Engine` by caching those particular diagnostics questions so that they never reach the `Diagnostics Engine` interface.

In addition to creating two messaging channels (one for the nurses and one for self-service patients), the team creates a new service called the `Diagnostics`

`Outbreak Cache Server` that handles all requests related to a particular outbreak or flu-related question (Figure 22-11). This new architecture reduces the number of calls to the diagnostics engine, allowing for more concurrent requests related to other symptoms. Without the risk-storming effort, this risk might not have been identified until flu season.

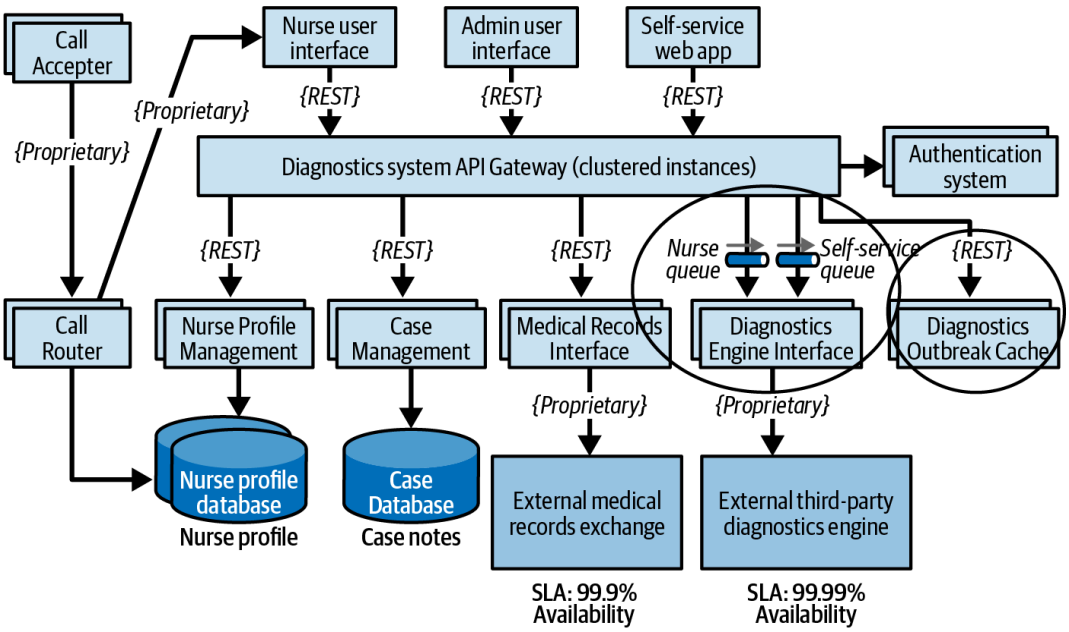

Figure 22-11. Architecture modifications to address elasticity risk

Security

Encouraged by these successes, the architect decides to facilitate a final risk-storming session focused on another crucial characteristic for this system—security. Due to HIPAA regulatory requirements, the `Medical Records Interface` must allow only nurses to access patients' medical records. The architect believes that the authentication and authorization checks in the API Gateway neutralize this risk, but is curious whether the participants will find any other security risks.

The participants all identify the diagnostics system's API Gateway as a high security risk (6). They cite the high impact if administrative staff or self-service patients access medical records (3), but rate the likelihood as medium (2). While the security checks for each API call help, all calls (self-service, admin, and nurses) are still going through the same API Gateway. They eventually convince the facilitator, who had initially rated this risk as low (2), that the risk is in fact high and needs to be mitigated.

Everyone agrees that having separate API Gateways for each type of user (admin staff, self-service users, and nurses) would prevent non-nurse calls from ever reaching the `Medical Records Interface`. The architect's final version of the architecture is shown in Figure 22-12.

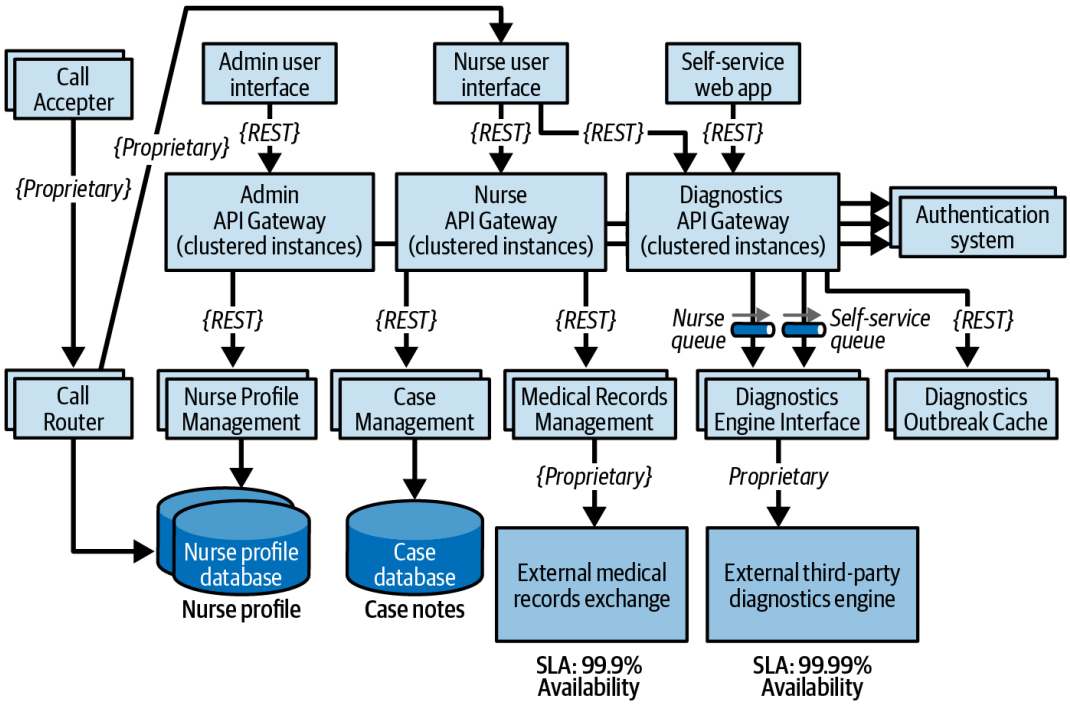

Figure 22-12. Architecture modifications to address security risk

This example illustrates the power of risk storming. Architects, developers, and key stakeholders collaborate, consider the architectural characteristics most vital to the system's success, and identify risk areas that would otherwise have gone unnoticed.

The original architecture (Figure 22-8) is significantly changed after risk storming (Figure 22-12) in ways that address concerns about availability, elasticity, and security and make this architecture more effective and more likely to succeed.

Summary

Risk storming is not a one-time process. It continues throughout a system's lifecycle, as teams work to identify and mitigate risk areas before they appear in production. How often an organization should hold risk-storming sessions depends on many factors, including the frequency of change, any architecture-refactoring efforts, and the architecture's incremental development. It's typical to do some risk storming on particular dimensions after adding a major feature or at the end of every iteration, to ensure that the architecture is correct and will address the business's needs.