CHAPTER 4
Design Tradeoffs

By Christoph Kern
with Brian Gustafson, Paul Blankinship, and Felix Gröbert

Security and reliability needs often seem difficult to reconcile with a project's feature and cost requirements. This chapter covers the importance of reviewing your system's security and reliability needs as early as possible in the software design phase.

We start by talking about the connection between system constraints and product features, then provide two examples—a payment processing service and a microservices framework—that demonstrate some common security and reliability tradeoffs. We conclude with a discussion about the natural tendency to defer security and reliability work, and how early investment in security and reliability can lead to sustained project velocity.

So you're going to build a (software) product! You'll have lots of things to think about in this complex journey from devising high-level plans to deploying code.

Typically, you'll start out with a rough idea of what the product or service is going to do. This might, for example, take the form of a high-level concept for a game, or a set of high-level business requirements for a cloud-based productivity application. You'll also develop high-level plans for how the service offering will be funded.

As you delve into the design process and your ideas about the shape of the product become more specific, additional requirements and constraints on the design and implementation of the application tend to emerge. There'll be specific requirements for the functionality of the product, and general constraints, such as development and operational costs. You'll also come upon requirements and constraints for security and reliability: your service will likely have certain availability and reliability

requirements, and you might have security requirements for protecting sensitive user data handled by your application.

Some of these requirements and constraints may be in conflict with each other, and you'll need to make tradeoffs and find the right balance between them.

Design Objectives and Requirements

The feature requirements for your product will tend to have significantly different characteristics than your requirements for security and reliability. Let's take a closer look at the types of requirements you'll face when designing a product.

Feature Requirements

Feature requirements, also known as *functional requirements*,[1] identify the primary function of a service or application and describe how a user can accomplish a particular task or satisfy a particular need. They are often expressed in terms of *use cases*, *user stories*, or *user journeys*—sequences of interactions between a user and the service or application. *Critical requirements* are the subset of feature requirements that are essential to the product or service. If a design does not satisfy a critical requirement or critical user story, you don't have a viable product.

Feature requirements are typically the primary drivers for your design decisions. After all, you're trying to build a system or service that satisfies a particular set of needs for the group of users you have in mind. You often have to make tradeoff decisions between the various requirements. With that in mind, it is useful to distinguish critical requirements from other feature requirements.

Usually, a number of requirements apply to the entire application or service. These requirements often don't show up in user stories or individual feature requirements. Instead, they're stated once in centralized requirements documentation, or even implicitly assumed. Here's an example:

All views/pages of the application's web UI must:

- Follow common visual design guidelines
- Adhere to accessibility guidelines
- Have a footer with links to privacy policy and ToS (Terms of Service)

1 For a more formal treatment, see The MITRE Systems Engineering Guide and ISO/IEC/IEEE 29148-2018(E).

Nonfunctional Requirements

Several categories of requirements focus on general attributes or behaviors of the system, rather than specific behaviors. These *nonfunctional requirements* are relevant to our focus—security and reliability. For example:

- What are the exclusive circumstances under which someone (an external user, customer-support agent, or operations engineer) may have access to certain data?
- What are the service level objectives (SLOs) for metrics such as uptime or 95th-percentile and 99th-percentile response latency? How does the system respond under load above a certain threshold?

When balancing requirements, it can be helpful to simultaneously consider requirements in areas beyond the system itself, since choices in those areas can have significant impact on core system requirements. Those broader areas include the following:

Development efficiency and velocity
: Given the chosen implementation language, application frameworks, testing processes, and build processes, how efficiently can developers iterate on new features? How efficiently can developers understand and modify or debug existing code?

Deployment velocity
: How long does it take from the time a feature is developed to the time this feature is available to users/customers?

Features Versus Emergent Properties

Feature requirements usually exhibit a fairly straightforward connection between the requirements, the code that satisfies those requirements, and tests that validate the implementation. For example:

Specification
: A user story or requirement might stipulate how a signed-in user of an application can view and modify the personal data associated with their user profile (such as their name and contact information).

Implementation
: A web or mobile application based on this specification would typically have code that very specifically relates to that requirement, such as the following:

- Structured types to represent the profile data
- UI code to present and permit modification of the profile data
- Server-side RPC or HTTP action handlers to query the signed-in user's profile data from a data store, and to accept updated information to be written to the data store

Validation
: Typically, there'd be an integration test that essentially walks through the specified user story step by step. The test might use a UI test driver to fill out and submit the "edit profile" form and then verify that the submitted data appears in the expected database record. There are likely also unit tests for individual steps in the user story.

In contrast, nonfunctional requirements—like reliability and security requirements—are often much more difficult to pin down. It would be nice if your web server had an `--enable_high_reliability_mode` flag, and to make your application reliable you'd simply need to flip that flag and pay your hosting or cloud provider a premium service fee. But there is no such flag, and no specific module or component in any application's source code that "implements" reliability.

Reliability and Security as Emergent Properties of System Design

Reliability is primarily an *emergent property* of the design of your system, and indeed the design of your entire development, deployment, and operations workflow. Reliability emerges from factors such as these:

- How your overall service is broken into components, such as microservices
- How your service's availability relates to the availability/reliability of its dependencies, including service backends, storage, and the underlying platform
- What mechanisms components use to communicate (such as RPCs, message queues, or event buses), how requests are routed, and how load balancing and load shedding are implemented and configured
- How unit testing, end-to-end functional testing, production readiness reviews (PRRs), load testing, and similar validation activities are integrated in your development and deployment workflow
- How the system is monitored, and whether available monitoring, metrics, and logs provide the information necessary to detect and respond to anomalies and failures

Similarly, the overall security posture of your service does not arise from a single "security module." Rather, it is an emergent property of many aspects of the way your system and operational environment are designed, including but not limited to these:

- How the larger system is decomposed into subcomponents, and the trust relationships between those components
- The implementation languages, platforms, and application/service frameworks on which the application is developed
- How security design and implementation reviews, security testing, and similar validation activities are integrated into your software development and deployment workflow
- The forms of security monitoring, audit logging, anomaly detection, and other tools that are available to your security analysts and incident responders

Finding the right balance among these many design objectives is difficult. Sound decisions often require a lot of experience, and even well-reasoned design decisions can turn out to be incorrect in hindsight. Chapter 7 discusses how to prepare for the inevitable need to revise and adapt.

Example: Google Design Document

Google uses a design document template to guide new feature design and to collect feedback from stakeholders before starting an engineering project.

The template sections pertaining to reliability and security considerations remind teams to think about the implications of their project and kick off the production readiness or security review processes if appropriate. Design reviews sometimes happen multiple quarters before engineers officially start thinking about the launch stage.

Google's Design Document Template

Here are the reliability- and security-related sections of the Google design document template:

Scalability
: How does your system scale? Consider both data size increase (if applicable) and traffic increase (if applicable).

  Consider the current hardware situation: adding more resources might take much longer than you think, or might be too expensive for your project. What initial resources will you need? You should plan for high utilization, but be aware that using more resources than you need will block expansion of your service.

Redundancy and reliability
: Discuss how the system will handle local data loss and transient errors (e.g., temporary outages), and how each affects your system.

Which systems or components require data backup? How is the data backed up? How is it restored? What happens between the time data is lost and the time it's restored?

In the case of a partial loss, can you keep serving? Can you restore only missing portions of your backups to your serving data store?

Dependency considerations
What happens if your dependencies on other services are unavailable for a period of time?

Which services must be running in order for your application to start? Don't forget subtle dependencies like resolving names using DNS or checking the local time.

Are you introducing any dependency cycles, such as blocking on a system that can't run if your application isn't already up? If you have doubts, discuss your use case with the team that owns the system you depend on.

Data integrity
How will you find out about data corruption or loss in your data stores?

What sources of data loss are detected (user error, application bugs, storage platform bugs, site/replica disasters)?

How long will it take to notice each of these types of losses?

What is your plan to recover from each of these types of losses?

SLA requirements
What mechanisms are in place for auditing and monitoring the service level guarantees of your application?

How can you guarantee the stated level of reliability?

Security and privacy considerations
Our systems get attacked regularly. Think about potential attacks relevant for this design and describe the worst-case impact it would have, along with the countermeasures you have in place to prevent or mitigate each attack.

List any known vulnerabilities or potentially insecure dependencies.

If, for some reason, your application doesn't have security or privacy considerations, explicitly state so and why.

Once your design document is finalized, file a quick security design review. The design review will help avoid systemic security issues that can delay or block your final security review.

Balancing Requirements

Because the attributes of a system that satisfy security and reliability concerns are largely emergent properties, they tend to interact both with implementations of feature requirements and with each other. As a result, it's particularly difficult to reason about tradeoffs involving security and reliability as a standalone topic.

Cost of Adding Reliability and Security to Existing Systems

The emergent nature of security and reliability means that design choices related to these considerations are often fairly fundamental, and similar in nature to basic architectural choices like whether to use a relational or NoSQL database for storage, or whether to use a monolithic or microservices architecture. It's usually difficult to "bolt on" security and reliability to an existing system that wasn't designed from the outset with these concerns in mind. If a system lacks well-defined and understandable interfaces between components and contains a tangled set of dependencies, it likely will have lower availability and be prone to bugs with security consequences (see Chapter 6). No amount of testing and tactical bug-fixing will change that.

Accommodating security and reliability requirements in an existing system often requires significant design changes, major refactorings, or even partial rewrites, and can become very expensive and time-consuming. Furthermore, such changes might have to be made under time pressure in response to a security or reliability incident—but making significant design changes to a deployed system in a hurry comes with a significant risk of introducing additional flaws. It's therefore important to consider security and reliability requirements and corresponding design tradeoffs from the early planning phases of a software project. These discussions should involve security and SRE teams, if your organization has them.

This section presents an example that illustrates the kinds of tradeoffs you might have to consider. Some parts of this example delve quite deeply into technical details, which aren't necessarily important in and of themselves. All of the compliance, regulatory, legal, and business considerations that go into designing payment processing systems and their operation aren't important for this example either. Instead, the purpose is to illustrate the complex interdependencies between requirements. In other words, the focus isn't on the nitty-gritty details about protecting credit card numbers, but rather the thought process that goes into designing a system with complex security and reliability requirements.

Example: Payment Processing

Imagine that you're building an online service that sells widgets to consumers.[2] The service's specification includes a user story stipulating that a user can pick widgets from an online catalog by using a mobile or web application. The user can then purchase the chosen widgets, which requires that they provide details for a payment method.

Security and reliability considerations

Accepting payment information introduces significant security and reliability considerations for the system's design and organizational processes. Names, addresses, and credit card numbers are sensitive personal data that require special safeguards[3] and can subject your system to regulatory standards, depending on the applicable jurisdiction. Accepting payment information may also bring the service in scope for compliance with industry-level or regulatory security standards such as PCI DSS.

A compromise of this sensitive user information, especially personally identifiable information (PII), can have serious consequences for the project and even the entire organization/company. You might lose the trust of your users and customers, and lose their business as a result. In recent years, legislatures have enacted laws and regulations placing potentially time-consuming and expensive obligations on companies affected by data breaches. Some companies have even gone entirely out of business because of a severe security incident, as noted in Chapter 1.

In certain scenarios, a higher-level tradeoff at the product design level might free the application from processing payments—for example, perhaps the product can be recast in an advertising-based or community-funded model. For the purposes of our example, we'll stick with the premise that accepting payments is a critical requirement.

2 For the purposes of the example, it's not relevant what exactly is being sold—a media outlet might require payments for articles, a mobility company might require payments for transportation, an online marketplace might enable the purchase of physical goods that are shipped to consumers, or a food-ordering service might facilitate the delivery of takeout orders from local restaurants.

3 See, for example, McCallister, Erika, Tim Grance, and Karen Scarfone. 2010. NIST Special Publication 800-122, "Guide to Protecting the Confidentiality of Personally Identifiable Information (PII)." *https://oreil.ly/T9G4D*.

Using a third-party service provider to handle sensitive data

Often, the best way to mitigate security concerns about sensitive data is to not hold that data in the first place (for more on this topic, see Chapter 5). You may be able to arrange for sensitive data to never pass through your systems, or at least design the systems to not persistently store the data.[4] You can choose from various commercial payment service APIs to integrate with the application, and offload handling of payment information, payment transactions, and related concerns (such as fraud countermeasures) to the vendor.

Benefits. Depending on the circumstances, using a payment service may reduce risk and the degree to which you need to build in-house expertise to address risks in this area, instead relying on the provider's expertise:

- Your systems no longer hold the sensitive data, reducing the risk that a vulnerability in your systems or processes could result in a data compromise. Of course, a compromise of the third-party vendor could still compromise your users' data.
- Depending on the specific circumstances and applicable requirements, your contractual and compliance obligations under payment industry security standards may be simplified.
- You don't have to build and maintain infrastructure to protect the data at rest in your system's data stores. This could eliminate a significant amount of development and ongoing operational effort.
- Many third-party payment providers offer countermeasures against fraudulent transactions and payment risk assessment services. You may be able to use these features to reduce your payment fraud risk, without having to build and maintain the underlying infrastructure yourself.

On the flip side, relying on a third-party service provider introduces costs and risks of its own.

Costs and nontechnical risks. Obviously, the provider will charge fees. Transaction volume will likely inform your choice here—beyond a certain volume, it's probably more cost-effective to process transactions in-house.

You also need to consider the engineering cost of relying on a third-party dependency: your team will have to learn how to use the vendor's API, and you might have to track changes/releases of the API on the vendor's schedule.

4 Note that whether or not this is appropriate may depend on regulatory frameworks your organization is subject to; these regulatory matters are outside the scope of this book.

Reliability risks. By outsourcing payment processing, you add an additional dependency to your application—in this case, a third-party service. Additional dependencies often introduce additional failure modes. In the case of third-party dependencies, these failure modes may be partially out of your control. For example, your user story "user can buy their chosen widgets" may fail if the payment provider's service is down or unreachable via the network. The significance of this risk depends on the payment provider's adherence to the SLAs that you have with that provider.

You might address this risk by introducing redundancy into the system (see Chapter 8)—in this case, by adding an alternate payment provider to which your service can fail over. This redundancy introduces cost and complexity—the two payment providers most likely have different APIs, so you must design your system to be able to talk to both, along with all the additional engineering and operational costs, plus increased exposure to bugs or security compromises.

You could also mitigate the reliability risk through fallback mechanisms on your side. For example, you might insert a queueing mechanism into the communication channel with the payment provider to buffer transaction data if the payment service is unreachable. Doing so would allow the "purchase flow" user story to proceed during a payment service outage.

However, adding the message queueing mechanism introduces extra complexity and may introduce its own failure modes. If the message queue is not designed to be reliable (for example, it stores data in volatile memory only), you can lose transactions—a new risk surface. More generally, subsystems that are exercised only in rare and exceptional circumstances can harbor hidden bugs and reliability issues.

You could choose to use a more reliable message queue implementation. This likely involves either an in-memory storage system that is distributed across multiple physical locations, again introducing complexity, or storage on persistent disk. Storing the data on disk, even if only in exceptional scenarios, reintroduces the concerns about storing sensitive data (risk of compromise, compliance considerations, etc.) that you were trying to avoid in the first place. In particular, some payment data is never even allowed to hit disk, which makes a retry queue that relies on persistent storage difficult to apply in this scenario.

In this light, you may have to consider attacks (in particular, attacks by insiders) that purposely break the link with the payment provider in order to activate local queueing of transaction data, which may then be compromised.

In summary, you end up encountering a security risk that arose from your attempt to mitigate a reliability risk, which in turn arose because you were trying to mitigate a security risk!

Security risks. The design choice to rely on a third-party service also raises immediate security considerations.

First, you're entrusting sensitive customer data to a third-party vendor. You'll want to choose a vendor whose security stance is at least equal to your own, and will have to carefully evaluate vendors during selection and on an ongoing basis. This is not an easy task, and there are complex contractual, regulatory, and liability considerations that are outside the scope of this book and which should be referred to your counsel.

Second, integrating with the vendor's service may require you to link a vendor-supplied library into your application. This introduces the risk that a vulnerability in that library, or one of its transitive dependencies, may result in a vulnerability in *your* systems. You may consider mitigating this risk by sandboxing the library[5] and by being prepared to quickly deploy updated versions of it (see Chapter 7). You can largely avoid this concern by using a vendor that does not require you to link a proprietary library into your service (see Chapter 6). Proprietary libraries can be avoided if the vendor exposes its API using an open protocol like REST+JSON, XML, SOAP, or gRPC.

You may need to include a JavaScript library in your web application client in order to integrate with the vendor. Doing so allows you to avoid passing payment data through your systems, even temporarily—instead, payment data can be sent from a user's browser directly to the provider's web service. However, this integration raises similar concerns as including a server-side library: the vendor's library code runs with full privileges in the web origin of your application.[6] A vulnerability in that code or a compromise of the server that's serving that library can lead to your application being compromised. You might consider mitigating that risk by sandboxing payment-related functionality in a separate web origin or sandboxed iframe. However, this tactic means that you need a secure cross-origin communications mechanism, again introducing complexity and additional failure modes. Alternatively, the payment vendor might offer an integration based on HTTP redirects, but this can result in a less smooth user experience.

Design choices related to nonfunctional requirements can have fairly far-reaching implications in areas of domain-specific technical expertise: we started out discussing a tradeoff related to mitigating risks associated with handling payment data, and ended up thinking about considerations that are deep in the realm of web platform security. Along the way, we also encountered contractual and regulatory concerns.

5 See, e.g., the Sandboxed API project.

6 For more on this subject, see Zalewski, Michał. 2011. *The Tangled Web: A Guide to Securing Modern Web Applications*. San Francisco, CA: No Starch Press.

Managing Tensions and Aligning Goals

With some up-front planning, you can often satisfy important nonfunctional requirements like security and reliability without having to give up features, and at reasonable cost. When stepping back to consider security and reliability in the context of the entire system and development and operations workflow, it often becomes apparent that these goals are very much aligned with general software quality attributes.

Example: Microservices and the Google Web Application Framework

Consider the evolution of a Google-internal framework for microservices and web applications. The primary goal of the team creating the framework was to streamline the development and operation of applications and services for large organizations. In designing this framework, the team incorporated the key idea of applying static and dynamic *conformance checks* to ensure that application code adheres to various coding guidelines and best practices. For example, a conformance check verifies that all values passed between concurrent execution contexts are of immutable types—a practice that drastically reduces the likelihood of concurrency bugs. Another set of conformance checks enforces isolation constraints between components, which makes it much less likely that a change in one component/module of the application results in a bug in another component.

Because applications built on this framework have a fairly rigid and well-defined structure, the framework can provide out-of-the-box automation for many common development and deployment tasks—from scaffolding for new components, to automated setup of continuous integration (CI) environments, to largely automated production deployments. These benefits have made this framework quite popular among Google developers.

What does all this have to do with security and reliability? The framework development team collaborated with SRE and security teams throughout the design and implementation phases, ensuring that security and reliability best practices were woven into the fabric of the framework—not just bolted on at the end. The framework takes responsibility for handling many common security and reliability concerns. Similarly, it automatically sets up monitoring for operational metrics and incorporates reliability features like health checking and SLA compliance.

For example, the framework's web application support handles most common types of web application vulnerabilities.[7] Through a combination of API design and code conformance checks, it effectively prevents developers from accidentally introducing

7 See, e.g., the OWASP Top 10 and CWE/SANS TOP 25 Most Dangerous Software Errors.

many common types of vulnerabilities in application code.[8] With respect to these types of vulnerabilities, the framework goes beyond "security by default"—rather, it takes full responsibility for security, and actively ensures that any application based on it is not affected by these risks. We discuss how this is accomplished in more detail in Chapters 6 and 12.

Reliability and Security Benefits of Software Development Frameworks

A robust and commonly used framework with built-in reliability and security features is a win-win scenario: developers adopt the framework because it simplifies application development and automates common chores, making their daily work easier and more productive. The framework provides a common feature surface where security engineers and SREs can build new functionality, creating opportunities for improved automation and accelerating overall project velocity.

At the same time, building on this framework results in inherently more secure and reliable systems, because the framework automatically takes care of common security and reliability concerns. It also makes security and production readiness reviews much more efficient: if a software project's continuous builds and tests are green (indicating that its code complies with framework-level conformance checks), you can be quite confident that it's not affected by the common security concerns already addressed by the framework. Similarly, by stopping releases when the error budget is consumed, the framework's deployment automation ensures that the service adheres to its SLA; see Chapter 16 in the SRE workbook. Security engineers and SREs can use their time to focus on more interesting design-level concerns. Finally, bug fixes and improvements in code that is part of a centrally maintained framework are automatically propagated to applications whenever they are rebuilt (with up-to-date dependencies) and deployed.[9]

Aligning Emergent-Property Requirements

The framework example illustrates that, contrary to common perception, security- and reliability-related goals are often well aligned with other product goals—especially code and project health and maintainability and long-term, sustained project

8 See Kern, Christoph. 2014. "Securing the Tangled Web." *Communications of the ACM* 57(9): 38–47. doi: 10.1145/2643134.

9 At Google, software is typically built from the HEAD of a common repository, which causes all dependencies to be updated automatically with every build. See Potvin, Rachel, and Josh Levenberg. 2016. "Why Google Stores Billions of Lines of Code in a Single Repository." *Communications of the ACM* 59(7): 78–87. *https://oreil.ly/jXTZM*.

velocity. In contrast, attempting to retrofit security and reliability goals as a late add-on often leads to increased risks and costs.

Priorities for security and reliability can also align with priorities in other areas:

- As discussed in Chapter 6, system design that enables people to effectively and accurately reason about invariants and behaviors of the system is crucial for security and reliability. Understandability is also a key code and project health attribute, and a key support for development velocity: an understandable system is easier to debug and to modify (without introducing bugs in the first place).
- Designing for recovery (see Chapter 9) allows us to quantify and control the risk introduced by changes and rollouts. Typically, the design principles discussed here support a higher rate of change (i.e., deployment velocity) than we could achieve otherwise.
- Security and reliability demand that we design for a changing landscape (see Chapter 7). Doing so makes our system design more adaptable and positions us not only to swiftly address newly emerging vulnerabilities and attack scenarios, but also to accommodate changing business requirements more quickly.

Initial Velocity Versus Sustained Velocity

There's a natural tendency, especially in smaller teams, to defer security and reliability concerns until some point in the future ("We'll add in security and worry about scaling after we have some customers"). Teams commonly justify ignoring security and reliability as early and primary design drivers for the sake of "velocity"—they're concerned that spending time thinking about and addressing these concerns will slow development and introduce unacceptable delays into their first release cycle.

It's important to make a distinction between *initial velocity* and *sustained velocity*. Choosing to not account for critical requirements like security, reliability, and maintainability early in the project cycle may indeed increase your project's velocity early in the project's lifetime. However, experience shows that doing so also usually *slows you down significantly* later.[10] The late-stage cost of retrofitting a design to accommodate requirements that manifest as emergent properties can be *very* substantial. Furthermore, making invasive late-stage changes to address security and reliability risks can in itself introduce even more security and reliability risks. Therefore, it's important to embed security and reliability in your team culture early on (for more on this topic, see Chapter 21).

10 See the discussion of tactical programming versus strategic programming in Ousterhout, John. 2018. *A Philosophy of Software Design*. Palo Alto, CA: Yaknyam Press. Martin Fowler makes similar observations.

The early history of the internet,[11] and the design and evolution of the underlying protocols such as IP, TCP, DNS, and BGP, offers an interesting perspective on this topic. Reliability—in particular, survivability of the network even in the face of outages of nodes[12] and reliability of communications despite failure-prone links[13]—were explicit and high-priority design goals of the early precursors of today's internet, such as ARPANET.

Security, however, is not mentioned much in early internet papers and documentation. Early networks were essentially closed, with nodes operated by trusted research and government institutions. But in today's open internet, this assumption does not hold at all—many types of malicious actors are participating in the network (see Chapter 2).

The internet's foundational protocols—IP, UDP, and TCP—have no provision to authenticate the originator of transmissions, nor to detect intentional, malicious modification of data by an intermediate node in the network. Many higher-level protocols, such as HTTP or DNS, are inherently vulnerable to various attacks by malicious participants in the network. Over time, secure protocols or protocol extensions have been developed to defend against such attacks. For example, HTTPS augments HTTP by transferring data over an authenticated, secure channel. At the IP layer, IPsec cryptographically authenticates network-level peers and provides data integrity and confidentiality. IPsec can be used to establish VPNs over untrusted IP networks.

However, widely deploying these secure protocols has proven to be rather difficult. We're now approximately 50 years into the internet's history, and significant commercial usage of the internet began perhaps 25 or 30 years ago—yet there is still a substantial fraction of web traffic that does not use HTTPS.[14]

For another example of the tradeoff between initial and sustained velocity (in this case from outside the security and reliability realm), consider Agile development processes. A primary goal of Agile development workflows is to increase development and deployment velocity—in particular, to reduce the latency between feature specification and deployment. However, Agile workflows typically rely on reasonably mature unit and integration testing practices and a solid continuous integration

11 See RFC 2235 and Leiner, Barry M. et al. 2009. "A Brief History of the Internet." *ACM SIGCOMM Computer Communication Review* 39(5): 22–31. doi:10.1145/1629607.1629613.

12 Baran, Paul. 1964. "On Distributed Communications Networks." *IEEE Transactions on Communications Systems* 12(1): 1–9. doi:10.1109/TCOM.1964.1088883.

13 Roberts, Lawrence G., and Barry D. Wessler. 1970. "Computer Network Development to Achieve Resource Sharing." *Proceedings of the 1970 Spring Joint Computing Conference*: 543–549. doi:10.1145/1476936.1477020.

14 Felt, Adrienne Porter, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. "Measuring HTTPS Adoption on the Web." *Proceedings of the 26th USENIX Conference on Security Symposium*: 1323–1338. *https://oreil.ly/G1A9q*.

infrastructure, which require an up-front investment to establish, in exchange for long-term benefits to velocity and stability.

More generally, you can choose to prioritize initial project velocity above all else—you can develop the first iteration of your web app without tests, and with a release process that amounts to copying tarballs to production hosts. You'll probably get your first demo out relatively quickly, but by your third release, your project will quite possibly be behind schedule and saddled with technical debt.

We've already touched on alignment between reliability and velocity: investing in a mature continuous integration/continuous deployment (CI/CD) workflow and infrastructure supports frequent production releases with a managed and acceptable reliability risk (see Chapter 7). But setting up such a workflow requires some up-front investment—for example, you will need the following:

- Unit and integration test coverage robust enough to ensure an acceptably low risk of defects for production releases, without requiring major human release qualification work
- A CI/CD pipeline that is itself reliable
- A frequently exercised, reliable infrastructure for staggered production rollouts and rollbacks
- A software architecture that permits decoupled rollouts of code and configurations (e.g., "feature flags")

This investment is typically modest when made early in a product's lifecycle, and it requires only incremental effort by developers to maintain good test coverage and "green builds" on an ongoing basis. In contrast, a development workflow with poor test automation, reliance on manual steps in deployment, and long release cycles tends to eventually bog down a project as it grows in complexity. At that point, retrofitting test and release automation tends to require a lot of work all at once and might slow down your project even more. Furthermore, tests retrofitted to a mature system can sometimes fall into the trap of exercising the current buggy behavior more than the correct, intended behavior.

These investments are beneficial for projects of all sizes. However, larger organizations can enjoy even more benefits of scale, as you can amortize the cost across many projects—an individual project's investment then boils down to a commitment to use centrally maintained frameworks and workflows.

When it comes to making security-focused design choices that contribute to sustained velocity, we recommend choosing a framework and workflow that provide secure-by-construction defense against relevant classes of vulnerabilities. This choice can drastically reduce, or even eliminate, the risk of introducing such vulnerabilities during ongoing development and maintenance of your application's codebase (see

Chapters 6 and 12). This commitment generally doesn't involve significant up-front investment—rather, it entails an incremental and typically modest ongoing effort to adhere to the framework's constraints. In return, you drastically reduce your risk of unplanned system outages or security response fire drills throwing deployment schedules into disarray. Additionally, your release-time security and production readiness reviews are much more likely to go smoothly.

Conclusion

It's not easy to design and build secure and reliable systems, especially since security and reliability are primarily emergent properties of the entire development and operations workflow. This undertaking involves thinking about a lot of rather complex topics, many of which at first don't seem all that related to addressing the primary feature requirements of your service.

Your design process will involve numerous tradeoffs between security, reliability, and feature requirements. In many cases, these tradeoffs will at first appear to be in direct conflict. It might seem tempting to avoid these issues in the early stages of a project and "deal with them later"—but doing so often comes at significant cost and risk to your project: once your service is live, reliability and security are not optional. If your service is down, you may lose business; and if your service is compromised, responding will require all hands on deck. But with good planning and careful design, it is often possible to satisfy all three of these aspects. What's more, you can do so with modest additional up-front cost, and often with a reduced total engineering effort over the lifetime of the system.