Release It! Design and Deploy Production-Ready Software

2nd edition (2018), Michael T. Nygard

1. + 2. Introduction

High-level learnings:

- You should not assume that you can plan for catching every kind of issue that could happen in production. There will always be some unpredictable event. The best you can do is to prevent those issues that are predictable, and build the system in a way that it can recover from any unanticipated event. (P. 1)
- Typically, systems spend more time in their "operation" phase than in the "development" phase of the SDLC. Software delivers its value in production (not during development). Decisions that you make regarding *design* and *architecture* influence the costs of both implementing and operating the software. In long-lasting, non-agile projects, if you tried to plan everything ahead of time (with little practical hands-on knowledge that can only be gained when operating software for some time), you would make many mistakes, because these early decisions are the least informed ones. Consequently, using an agile development approach is the suggested way to go: build small increments that you also *operate* right from the beginning. Not just to learn which features the customer really needs (via feedback), but also to learn about real-world production issues. (P. 4)

3. System stability

- From experience, the authors state that making design decisions that result in a stable system usually costs the same as implementing an *un*stable system. (P. 24)
- Definition of terminology: (P. 24/25)
  - **Transaction**: abstract unit of work processed by the *system*. E.g. "customer orders several items". Thus, it can comprise more than just a single DB transaction.
  - **System**: the conglomerate of different hardware and software that is required to process transactions.
  - **Robustness / stability**: ability of a system to keep processing transactions even when *impulses* or *stresses* are affecting it, or when one or more components fail.
  - **Impulse**: a fast change in load applied to the system, e.g. a large chunk of incoming requests, or a DoS attack.
  - **Stress**: a persistent (long-lasting) force applied to the system, e.g. a component that is nearing its capacity, or external systems that keep responding very slowly.
- A typical category of production incidents are only triggered when operating a system for *long* periods of time. Examples are memory leaks, or the system slowing down due to having stored excessive amounts of data. Automated *tests* in CI don't catch those

issues, and neither do load tests or stress tests. They are not running long enough. Running load tests for long periods of time would be too costly anyway. What you can do, though, is to set up a separate system (e.g. a smaller replica of the production system) that you operate for longer time periods (while not updating the software - so *not* a typical “staging” environment), and to which you apply typical production-loads - including the typical variadic day-and-night load patterns. (P. 26)

- The book defines the term “failure mode” (but defines it differently than other sources on the internet): in this book, “failure mode” is defined as: the original problem (the “fault”) that occurred in some component + how this problem propagates through the remaining system + the resulting damage. (P. 26)
- Definition of terms (P. 28/29):
    - **Fault**: the cause of a problem, e.g. a bug in the code, e.g. when forgetting to check a rare condition. The effect might not be immediately visible yet.
    - **Error**: the *visible* incorrect effect of the *fault*. The wrong things the system does because of the fault.
    - **Failure**: the final bad state of your system, e.g. being unresponsive.
    - Generally, faults become errors, errors provoke failures.
- The high-level mission of this book is to teach us to build *safe* failure modes, where the damage of a fault is contained to a small part of the system (i.e., we avoid that the problem spreads throughout the entire system). To achieve this, you need to be aware of the different faults that can occur, how they would spread through the system, and how you can steer / control this spread. (P. 27)
    - Examples are:
        - Using timeouts instead of calling methods that block for indefinite time periods. A blocking method can cause resource starvation in the system calling that method.
        - Not using something like Java RMI that hides the "remoteness" of calls and can cause the calling system to hang.
- A general insight is that *tight coupling* between two system-components facilitates the spreading (or even multiplication) of an issue between these two components. (P. 29)
- There are two “camps” when it comes to how to handle faults. One camp says that you need to try as hard as possible to be “fault-tolerant”, i.e., think of every kind of *fault* that can happen, and develop a lot of defensive code that prevents such *faults* from becoming *errors*. The other camp states that this approach is futile, because you cannot possibly find every kind of fault anyway. They instead say: “let the system crash and restart it from a known good state”. (P. 29)
    - For instance, if you started by asking “what are all the things that can go wrong when calling external/internal components, using resources, etc.”, you would get a very long list of questions. You would have to collect all the resources you are using/calling, and for each one you would have to assume various faults, such as “resource is not available (anymore)”, “resource is slow”, “resource is full”, …

4. Stability antipatterns

- From the (uncountably) many concrete faults and causal effects (errors and failures) that there are, it is possible to bundle/categorize them into a small set of patterns, which are presented in this section.
- Term definition: “integration point”: the interaction points between two systems, e.g. a TCP socket, a UNIX pipe, a RPC, etc. - all these have the ability to be slow, or hang, or deliver incorrect outputs (P. 35). The book goes into a few details about different kinds of integration points:
    - **Sockets**:
        - Establishing a socket can fail fast (e.g. when you try to connect to a closed port, causing a TCP *reset* packet, or if the port is open but its TCP listening queue is already full), or fail very slowly (e.g. when a port is open, and your connection request ends in its (not yet full) listening queue). (P. 37)
        - *Default* values for the connection timeouts vary between OSes, and they could be several minutes!
        - After a connection has been successfully established, it could still happen that data is not no longer quickly received or sent. (P. 37)
        - Be careful with long-lasting connections: they might die because of firewalls of the intermediate routers that drop connections from their NAT tables in case no packets were sent using those connections for a long time. In this case, the firewalls drop the routing table entries due to some TTL settings. Such dead connections are not quickly terminated by a TCP reset, because the firewalls might simply drop the packet. Sending regular heartbeats can be a way to circumvent the problem. (P. 41/42)
        - **Mitigation**: set a connection timeout and a socket timeout (covering the case when sending/receiving data is blocked).
        - If you want to debug networking issues, it can make sense to use tools such as tcpdump on the server to capture packets in promiscuous mode, then transfer the dump to your developer laptop and look at it e.g. with Wireshark. (P. 38)
    - **HTTP**:
        - Many issues can occur, such as (P. 43)
            - Server might establish the TCP connection, but not reply with the expected HTTP syntax (headers, then body)
            - The server might not read the packets you are sending to them, causing the socket write ops to block on the client side.
            - The server might send back unexpected stuff, e.g. invalid HTTP status codes, or stuff with unexpected Content-Type, or it might lie about the Content-Type.
        - **Mitigation**: configure timeouts, be prepared for any kind of invalid response.
    - **Vendor libraries** (P. 44)
        - While enterprise-grade *server* software (whose license you buy) often has

        high quality and are well-tested, the *client SDKs/libraries* that the vendor offers are often of much lower quality, using all kinds of unsafe coding practices.
        - The issues with the libraries might be bugs, or offering too little control (such as for the timeouts).
        - The libraries are often closed-source, making it very difficult to debug them.
        - Even if you report bugs, it may take a long time until these are fixed.
- The following is a list of stability antipatterns.
- **Chain reactions**: (P. 47)
    - Scenario: you have a horizontally-scaling system where a load balancer distributes load onto many nodes. One node fails due to being overloaded. That node's failure then causes a chain reaction causing other nodes to also fail (quickly), basically for the same reason (they have to pick up even more load).
    - If the fault that caused the first node to go down is really a software bug (and not simply a huge amount of load), then this bug is present in the code of all nodes, causing the same fault in those.
        - Such software bugs might be resource leaks (e.g. memory, sockets, connections from a pool, database (row) locks, …) or weird timing bugs (e.g. causing deadlocks).
    - Mitigation: autoscaling and health checks. However, the scaler needs to be fast enough (faster than the chain reaction itself).
- **Cascading failures** (P. 49/50):
    - When a failure in one part / layer / component of the system causes a failure in *another* part / layer / component.
    - An amplification factor is if you have many components that depend on *one* particular component: when that one fails, the failure propagates to all those components that depend on it.
    - Apart from complete failures, slow responses of the callee can also cause failures in the caller, if the caller is programmed to poorly hammer the callee with requests.
    - Mitigation: use Circuit Breakers, and Timeouts
- **Requests made by users:**
    - Term definition "system capacity": the maximum throughput your system can produce that is still satisfying to your users (performance is acceptable). (P. 52)
    - There are many system-internal limits that can be exhausted:
        - Memory: Swap memory is your enemy (it slows down the system). Also, be careful with keeping user data in memory for longer periods of time (e.g. session data). A surge of requests would kill your node, especially if it doesn't check for memory limits. Consider moving data to a different process that is optimized to handle large loads and manage memory without errors, e.g. Redis. (P. 54)
            - Some tips for caching (P. 67): configure the maximum available amount of memory, monitor cache *hit rate* (send alerts if it is too low). When implementing caches in your own code, use *weak*

      references (if offered by the programming language), so that the runtime's garbage collector can delete such cached objects. Implement some cache item invalidation strategy, and be careful with implementing "cache warming" phases, because they could cause the "database dogpile" antipattern (described below).

    - Sockets: Per IP, there are only 64511 ports available for connections. For a server to be able to accept more, you create virtual IPs. But your application needs to listen to all of them. And you need to know how to tweak the TCP parameters in the kernel and be aware of the memory needed by kernel buffers. There are also issues with *closed* sockets, which cannot be *immediately* reused for incoming connections, but depend on TIME_WAIT.
  - There are different kinds of users:
    - Expensive users (P. 56): those (paying) users that really make intensive use of your system. There is no real "defense" against them - you want those users, because they generate revenue. However, you can create automated tests that contain the most expensive *observed* request patterns, and make sure that the tests also pass even when doubling or quadrupling the load.
    - Unwanted or malicious users (P. 59), e.g. scrapers or botnets. The best you can do is to detect and block them on the network level.
    - Self-denial attacks (P. 69): when your own employees (e.g. from the marketing department) are responsible for traffic surges, e.g. because of a promotional offer that has spread to way more users that you originally intended. Mitigations:
      - Auto-scaling, as long as it reacts fast enough. If you already know/ expect the date and time of a self-inflicted traffic surge, you can apply *pre*-scaling of your resources.
      - Improve the communication within your organization.
      - Don't distribute "deep links" where a surge of requests can bring down your system. Instead, distribute links to statically-served websites/landing-zones (e.g. hosted on a CDN) that redirect users to the actual sites, and which you can also "close down" once a promotion/offer has ended.
- **Scaling effects** (P. 71): when many services or components depend on one (or few) components. Such kinds of systems cannot scale properly. The worst design is if you have only one service instance, which is a SPOF (Single Point of Failure). Stress tests help to discover them.
- **Blocked threads** (P. 63):
  - Often, some kind of blocked threads (that wait for an impossible outcome) are the cause of a system *failure* (the system still runs, but is unresponsive).
  - Mitigations:
    - Specify timeouts wherever possible.
    - Expose (and monitor) metrics (e.g. counters of resources (e.g. open connections, or # of running worker-processes) or status codes of

        requests).
        - Do external monitoring where a mock client (situated in a different data center) runs dummy queries against your system - it generates alerts sent to you once these queries fail.
        - When developing, carefully craft your code, and use well-known concurrency patterns with well-tested libraries. Beware of hidden complexity and waiting times that are not immediately obvious from looking at the API of things you are calling. E.g. when using a vendor's SDK that internally handles connection pools poorly.
- **Unbalanced capacities** (P. 75-78)
    - When the scaling-capabilities of the different services/components of your system do not match. The consequence is that one component can overload another (less scalable) component. *Unbalanced capacities* is a special form of the *Scaling Effects* antipattern.
    - One issue often found in practice is the reliance on external, *rate-limited* APIs, which in turn limit the scalability of *your* service. Mitigations: Circuit breaker pattern, or your (overloaded) service could also communicate a back-off time (applying back-pressure) to the caller.
    - Mitigations:
        - Capacity planning: have an idea how much different components need to scale (and possibly scale in different ways) depending on the overall system load. Also, take a close look at limiting "per-instance" resources, such as memory, sockets or threads.
        - Auto-scaling
        - Automated stress tests, which verify two things: 1) scaling of your application works as expected (or, if you are not using auto-scaling, the response time (or error rate) increases as expected). 2) Once the stressing calls stop, the system actually *recovers*: after a short recovery wait time, it serves requests normally again, with low response times and error rates.
- **Dogpile** - as in "a pack of dogs" (P. 78)
    - A temporary load surge caused by a bunch of components ("dogs") putting load onto another component all at the same time. Typical examples are
        - (Mass) "server startup", e.g. because of an outage (power cycle) or because of a scheduled update: they could cause problems with electricity, or put stress onto another component (such as the DB) because the starting servers are warming up their local cache (pulling data from the DB).
        - Too many Periodic (cron) jobs triggering at once
    - Mitigations:
        - For cron jobs, use a random clock slew to artificially delay jobs, spreading the load.
        - Configure the orchestration layer to not run too many jobs (e.g. start servers) in parallel.
- **Force Multiplier** (P. 80)

    - When fully-automated *control plane* software (managing infrastructure or your deployments)...
        - …makes mistakes (e.g. incorrectly detecting the liveliness of components, thus unnecessarily starting or killing components)
        - …amplifies mistakes (e.g. pushing a bad human-crafted configuration into all corners of your system)
        - …is in conflict with manual admin-activity (e.g. when an admin disables some components on purpose during a system migration, which the control plane incorrectly enables again, causing a system outage).
    - The mitigation is to carefully tune the control plane settings:
        - If the control plane thinks that almost everything is down, it might be an issue in the control plane itself (e.g. when it suffers from a network partition). -> escalate to a human
        - Use hysteresis: start machines quickly, but shut them down slowly. Starting is safer than shutting down.
        - When there is a *huge* drift between desired and actual state, escalate to a human. The detection of the actual state might simply be off.
        - For tools used by humans interactively, implement several kinds of upper / lower limits that keep the human operator "in check", warning them about unsafe operations. E.g. don't spin up an *infinite* number of instances of your components, or don't allow the number of instances to go below a certain threshold.
        - Beware of lag / momentum in your control loop: actions initiated by the control plane take time until they are detected by the control plane, so it should wait a considerable amount of time before scheduling or killing instances.
- **Slow responses** (P. 84-86):
    - When a component you call is slow, this can cause *Cascading failures* because the upstream component will also become slow and become vulnerable to stability problems. Slow responses tie up resources in the caller and callee component for longer time periods.
    - For websites (accessed by humans), slow responses lead to even more traffic, because users will start clicking the refresh button.
    - Mitigation:
        - Make up a *Service Level Agreement* for your service, as in "my service can deliver a response within X milliseconds, or it fails".
        - Write load tests that check how your system behaves if the latency to external services increases significantly.
- **Huge result data sets** (P. 86):
    - While the result data sets of API or DB calls are small during *development*, you might get an (unexpectedly) *huge* result set as response in a *production* setting where data has accumulated for a while. This typically causes your system to slow down, or even crash.
    - Mitigation:
        - When making queries, *always* use some sort of (explicitly-configured)

        "paging" feature to limit the maximum response size. Don't assume that the response size is small. Don't assume that, when omitting paging-related arguments, the destination keeps its "default behavior" - its behavior might change unexpectedly over time.
        - Be careful with ORMs: you might be able to limit the size of the original query, but you also need to *explicitly* limit queries when following an object's *associations*.

5. Stability patterns

The following is a list of stability patterns:

- **Timeouts** (P. 92):
    - Timeouts avoid the *cascading failures* antipattern, by making sure that calls to other services do not block the calling thread.
    - Apply timeouts to all kinds of places in your code, such as:
        - Resource pools
        - Concurrency primitives offered by your programming language (e.g. semaphores, mutexes)
    - Think about extracting/modularizing the logic that deals with calling functionality with timeouts and handling timeout-related exceptions. This is much better than having similar timeout-handling code *copied* all over your code base. The handler-logic of this centralized code could also include a Circuit Breaker.
    - Consider *queueing* failed calls so that you retry them later
- **Circuit Breaker** (P. 96)
    - The Circuit Breaker (CB) pattern handles failures of unreliable external services. It is like a **decorator** put in front of these services, which may fail randomly, e.g. in front of a DB server. It avoids antipatterns such as *cascading failures*, *unbalanced capacities*, or *slow responses*.
    - Its purpose is to avoid unnecessary **overloading** of a component that the CB already determined has failed, and to avoid that the calling (client) system itself may also get overloaded to trying too many repeated requests on the failed component. The basic assumption (which typically holds in practice) is that it makes little sense to *quickly* retry calling other failing services, because that other service would simply fail again (usually for the same reason).
    - When errors occur when calling the external service, these errors are returned to the caller, but internally the CB counts the number of consecutive failures. If the number of failures exceeds some threshold, the CB "trips" and from now on, new requests made by the caller are immediately answered by the CB with a special CircuitBreaker error. i.e. the CB does not even try to use the service anymore (or it may try to switch to an alternative service).
    - The CB pattern is typically implemented using the *decorator* pattern, which internally uses a *state-machine*, with these states:
        - **Closed:** underlying service works, calls are forwarded - this is the *initial* state

        - **Open:** the “tripped” state, on a call, the call is not forwarded to the underlying service, the CB immediately returns/throws a CB error. After a time threshold has been exceeded (between the last failed call, and the call now), the CB switches to Half-Open.
        - **Half-Open:** intermediate state (between Open → Closed) the CB is in while trying to re-establish the connection. On a call, the CB forwards it to the underlying component. If it fails again, immediately switch back to Open. If it succeeds, switch to Closed.
        - **Half-Closed:** intermediate state (between Closed → Open) the CB is in after the first failure occurred, but the CB still forwards calls to the underlying component, with the hope that it will recover (in which case the CB goes to Closed again)
    - Some implementation details:
        - The normal variant of the CB pattern is "synchronous", i.e. the CB only is active when the client calls methods on the underlying service the CB decorates. But there are also asynchronous variants where the CB has an internal background thread which tries to reestablish the connectivity to the service while in Half-Open state.
        - The CB may deal with different error types differently, e.g. have varying time thresholds for different errors (e.g. handle “connection refused” errors different than “response timeout” errors).
        - The CB may have a fallback strategy, e.g. returning a cached value, or returning a generic response rather than a personalized/specific one. Or even call a secondary service if the primary one has failed. As this kind of fallback-behavior has an impact on the business, you should involve all stakeholders when deciding how to handle failed calls.
        - Don’t just count the number of failures to decide when to switch from closed to open. Instead, count the *failure rate*, e.g. using the Leaky Bucket algorithm.
        - Add observability to the CB: log the state changes, offer metrics, track the frequency of state changes, and do monitoring on too high frequency values. It can also make sense to allow ops people to explicitly trip or reset the CB from outside, e.g. via an API.
        - Have one CB instance per (operating system) process. It would not be economical to have a “central” CB shared by multiple processes, as this would introduce a new failure mode: the IPC to the CB, the CB itself could fail, etc.
        - Use well-tested open source CB libraries whenever possible, to avoid that you make e.g. thread synchronization mistakes when rolling your own implementation.
- **Bulkheads** (P. 98)
    - When considering the analogy to a ship, bulkheads are the compartments that you can willingly seal off and “sacrifice” by flooding them with water, in order to contain damage to those flooded compartments.
    - The most typical form of bulkheads is redundancy: have multiple servers,

multiple instances of your components, and distribute your component instances to different servers. A load balancer then distributes the calls to the instances/servers. This way, your system does not collapse in case individual servers or individual services/components fail.
        - Another variant of bulkheads (at a finer granularity than the machines) is to pin applications to specific CPU cores. This avoids that a hungry multi-threading application can bring down an entire machine.
    - Consider reserving some resources (e.g. threads handling requests) to exclusively handle dedicated functionality, e.g. admin access. This lets you, say, collect data for postmortem analysis, or request an orderly shutdown.
- **Steady State** (P. 101)
    - Basic idea: keep the usage of state-related resources "steady" (at a constant level), by making sure that you have some *automated* mechanism in place that *releases* resources as needed, for every resource that is continuously allocated.
    - Examples for such state-related resources are disk space or memory.
    - The mechanism that releases resources should *not* be a human, because whenever humans fiddle around in your system, there is a chance for them to introduce errors (e.g. misconfigurations).
    - Use monitoring, e.g. measuring the memory or storage consumption, or the disk I/O rate (comparing it to what the disk can offer) of your database.
    - Purging user-data is often very complicated. You need to make sure not to break referential integrity of your data (e.g. avoiding orphans (deleting too little), or broken references (deleting too much)). You also need to ensure that the components keep working after the data has been purged, e.g. using automated tests.
    - Log files are another common source of problems, as they can grow unboundedly. Ensure that you have either configured log rotation, or actually ship logs to dedicated log-storage systems as quickly as possible.
- **Fail fast** (P. 106)
    - Whenever a called component can determine that the called operation will fail, it should return an error as quickly as possible, to avoid unnecessary allocation of resources in the callee and caller.
    - A common approach is to check (and pre-allocate) all necessary resources right at the beginning of an operation. For instance, the called system might require two external services. It makes sense to have pre-allocated clients for these services, and immediately fail when only a single one of these clients is not ready. If you are using CircuitBreakers, first check all of their states, before proceeding to call the underlying services. The basic idea is that we want to avoid that our component needs to call external service A (which works), but then discovers that service B is not available, and thus our component must fail (thus it fails, but fails slowly - the worst kind of failure).
    - Also, limit the request size. For instance, offer *paging* and validate the upper limit of the number of requested items. E.g. don't allow the caller to request a list of 9999999999999999 customers.
    - In general, even before pre-allocating resources, do input sanity checking and fail

early if there is a problem.
    - When failing, make sure to return the appropriate error codes / messages, e.g. differentiating “resource not available” errors from other application-level errors, such as “invalid parameters”. Only then can the caller react correctly (e.g. showing the correct message to the user in the frontend, or tripping the Circuit Breaker).
- **Let it crash** (P. 108)
    - The basic idea is: since writing defensive code to *recover* from errors is very difficult and unreliable, we instead crash and restart from a clean state as quickly as possible.
    - Make sure to choose a fine “crash granularity”. For instance, in a containerized microservice, it makes sense to only let the container crash and restart it, and not restart the entire container *runtime* or the entire machine.
    - The “Let it crash” approach only makes sense if these requirements hold:
        - You have some supervisor in place that detects crashes and auto-restarts the crashed components.
        - The start-up time of your component is fast. For instance, slow application servers that have long start and warm-up times are not suitable.
- **Handshaking** (P. 111)
    - In a general sense, “handshaking” refers to the first phase of two (virtual) devices communicating with each other, negotiating things. For instance, in serial protocols the receiver communicates how fast it wants to receive data, the data encoding, or when it is ready.
    - Handshaking is very common in lower-level protocols, but is almost nonexistent in high-level application protocols.
    - One particularly interesting aspect that is negotiated in handshaking is *flow control* a.k.a. *throttling*. We want our components to be able to protect themselves from being overloaded by the caller. One possible approximation of this is to use load balancers who are aware of the health of the services that they are forwarding the requests to. For instance, the health check of a service could indicate that it is overloaded by returning a HTTP 503 status code, indicating a *temporary* problem.
    - The *Circuit Breaker* pattern is like a poor-man’s version of *Handshaking*, as it introduces a throttling mechanism for components that don’t support it themselves.
- **Test Harness** (P. 113)
    - Basic idea: set up faulty instances of components that emulate remote systems, doing wrong things on purpose (which can happen in practice), especially on the network level.
    - Examples include:
        - Refuse connections straight away
        - Place incoming connections in the TCP socket listening queue and let them starve there.
        - Let the TCP connection establish, but never send any data
        - Send RESET packets at random points of time

        - Report a full *receive* window, never draining the data (thus blocking the write stream of the caller)
        - Simulate packet loss
        - Simulate low data rates, including extremely slow response rates (e.g. 1 byte every 30 seconds)
        - For HTTP/REST APIs:
            - Send response headers, but don't send the body
            - Send data in a different format than expected (e.g. HTML instead of JSON)
            - Send much more data than expected (gigabytes instead of kilobytes)
    - This is in contrast to "mock objects" whose purpose is to reproduce the *expected* behavior, conforming to the predefined specifications and interfaces.
    - It makes sense to use configurable proxy servers that let you inject such kinds of faulty behavior, such as limiting bandwidth, or adding jitter / latency. Tooling involves https://github.com/danielwellman/bane/ or https://github.com/Shopify/toxiproxy or https://github.com/tylertreat/comcast or https://github.com/sitespeedio/throttle
- **Decouple synchronous communication** (P. 117)
    - The basic idea is to consider *event-based* architectures. They use asynchronous message brokers, eliminating many failure modes that synchronous calls have.
- **Governor** (P. 123)
    - Basic idea: if you use some kind of control plane that automatically performs reactive actions (e.g. a node autoscaler), the governor is an additional (integrated) algorithm or component that ensures that "dangerous" / "costly" actions are slowed down enough for humans to get involved (alerting them).
    - For instance, the *governor* would ensure that you are not spending a huge cloud bill because the autoscaler decided to (incorrectly) scale your node pool up to 10'000 nodes.
    - It often makes sense to design the governor's algorithm to be *dynamic*: for instance, the algorithm of a node autoscaler could slow down the rate of adding more nodes only once there are already many nodes. In other words, you are separating the actions into "safe" and "unsafe" ranges, and only do alerting within the unsafe range.

6. Case study

Here the authors just tell a story of a production incident.

My only learning was on P. 134: if you collect metrics on the duration of requests, the metrics aggregation system (e.g. Prometheus) will only show those requests that actually have completed. In other words: requests that *failed* with a timeout won't be shown.

7. Networking, (virtual) machines and containers

- The concept "design for production" is introduced. It means that you need to think of production issues carefully. Aspects include: (P. 141)
    - Networking (production networks work very different to those in test or development environments)
    - You need observability features
    - Distributed systems need orchestration of services (and their deployments)
    - You need to take care of security
    - Treating "ops" people as users too -> try to make their life easy
- NICs and DNS (P. 143): the admin of a machine configures a machine's *hostname*. Together with its (search) domain, you get a fully-qualified domain name (FQDN), e.g. "server1.somecompany". In addition, there is the DNS server, which may assign completely different "hostnames" to the machines. The problem is that many developers (and applications) assume that the IPs resolved from a host's FQDN are the same IPs that are resolved from DNS queries, which is usually *not* the case in production, where machines have *multiple* NICs (with different IPs), which is also called "multi-homing".
    - Multi-homing has several purposes, e.g. separating production traffic from administrative traffic, such as monitoring or backup traffic.
    - When you start your own server, make sure to bind the listening socket to the right IP, or bind it to *all* IPs. The book seems to recommend against binding all interfaces (see P. 145), but I would do so, to facilitate debugging a production incident, e.g. from an administrative NIC.
- There are many different deployment options for your applications / services: (P. 146-153)
    - On a physical host - but you should avoid doing so, as the other options have a better resiliency (when you manage them with a control plane).
    - On a Virtual Machine. Be aware of caveats, such as overprovisioning done on the hypervisor level (e.g. allocating more virtual CPUs than physical cores are available), which leads to random/unexpected performance problems due to throttling (especially with "noisy neighbors"). The guest OS is unaware of these throttles happening.
    - In containers (Docker, etc.): be sure to adhere to containerization best practices, e.g. the 12-factor app rules.
- If VMs or containers are run in the *cloud* (vs. running them in a data center), be prepared that VMs and containers are *very* ephemeral / temporary, as the cloud's control plane could terminate them even more often than you would think, sometimes without reasoning.

8. Processes on machines

- For a reliable system, you need to make sure that your application's processes (running on the different machines) are running the right code and that they are configured correctly. (P. 155)

- There is a lot of *overloaded* terminology which is interpreted differently by different people. For instance, what does “reboot the server” mean? Does “server” in this context mean the server (Linux) *process*, or the entire server *machine*? Make sure to communicate with precision within your team and across teams. (P. 157)
- Regarding “running the right code”: (P. 158)
    - Always use a version control system
    - Be cautious about your supply chain: consider not downloading parts of the build tool chain (e.g. compiler binaries, Docker images, etc.) from semi/untrusted Internet sources, because someone could silently inject malware. Also beware of *plugins* of your build toolchain.
    - Always build production builds in CI, not on (often) polluted developer machines. Make sure that the repository that stores your application builds is secured, such that only the CI system can push to it.
    - Treat infrastructure and environments as immutable. E.g. Docker images are immutable: you take a “well-known” base image, apply some patches to it at build time, but then during production, you no longer modify them. The (bad) opposite approach would be to keep patching software during production over long time periods, which leaves you with hard-to-reproduce system states.
- Regarding *configuration* (P. 161):
    - Don’t store unencrypted configuration data in a repository where everyone can see it. Use different approaches, e.g. storing encrypted configuration data, or store it in a separate repository which only select people have access to.
    - Configuration can be distributed in many forms: it should always be deployed separately to the application (i.e., do not hard-code configuration into your application). E.g. as environment variables, or files. For distributed systems, the orchestration engine / control plane needs to assist you with customizing the configuration values for the different *instances* of your services.
- Observability (called “transparency” in the book): the book describes (using convoluted wording) that you need to add *metrics* and *logging* to your application.
    - Regarding *logging*: consider borrowing the “correlation ID” concept from *tracing*: add the correlation ID to your log messages. This makes it easier to understand which log messages refer to e.g. the same request. (P. 169)
    - Regarding *health checks*: consider adding more information, such as the application version, whether the instance is still accepting work (as in “Kubernetes *readiness* probe”), and the status of the internally-used resources, such as caches, circuit breakers, or connection pools. (P. 170)
        - Of course all this information could instead be exposed as *metrics*.

9. Networking

- This chapter is about networking-related aspects, such as routing, load balancing, failover, traffic management and service discovery (P. 171)
- There are several discovery mechanisms (that usually also do load balancing, when grouping multiple instances of a specific service type). Which one is most suitable for

you depends on your required rate of change (how often do services come and go). Examples are:

- Static configuration: you manually register service instances somewhere. Only makes sense if you have a very low rate of change. (P. 172)
- DNS: easy to configure, but requires manual updating. Load balancing is typically done in a “round robin” style, which is not load-aware. There is no health checking. With DNS you can achieve *Global Server Load Balancing* (GSLB) where DNS is used at the top of the routing hierarchy, returning IPs of *geographically-distributed* load balancers. A GSLB also keeps track of the health and responsiveness of the load balancers. (P. 173-175)
    - You should generally use separate infrastructure / servers for DNS itself, rather than the servers that run your application / system.

- Load balancers (P. 178+)
    - They forward requests to one or more pools of service instances.
    - Have various configuration options, such as:
        - Health checking approaches
        - Behavior when there are no healthy instances left
        - Load-balancing algorithm to use
        - Caching behavior
    - Can be implemented in software or hardware. Hardware load balancers have better throughput and can handle more parallel connections, but are pricey (expect five digits for low-end configurations, six digits for high-end configurations). (P. 180)
- Characteristics of overloaded systems (P. 182+)
    - Systems that are overloaded from high demand usually fail because of queues filling up in different places. Each request consumes a socket on each tier of your application (e.g. when the client calls service A which internally calls service B, then sockets are consumed by the client, A and B for that single request).
    - The number of available sockets on a server is limited, and the longer the duration of requests is, the lower the request throughput becomes. This leads to an unfortunate, *non-linear* relationship between incoming requests and (successfully) handled requests: the higher the rate of incoming requests, the more your servers are under load, thus the slower the processing of each request becomes, and thus the number of successfully processed requests decreases. And finally, due to users being impatient, users fire even more requests at slow systems, making them even slower.
    - Apart from sockets, you also need to keep an eye on network I/O bandwidth, which is also limited.
- Mitigation approach to avoid overloaded systems: **fail fast**, as early in your call chain as possible - the book refers to this as “load shedding” (P. 184)
    - The earlier you reject a request in your service call chain, the better - the best approach is to terminate requests right at the ingress / reverse proxy / load balancer. This avoids that resources are bound on several tiers before rejecting the request.
    - Technique #1: configure your services to have a short TCP listening queue

(many programming languages and server-frameworks allow you to configure this for the server socket, see e.g. the *backlog* argument for [creating a Python server](#))
    - Technique #2: add code to your service that immediately creates inbound sockets, but places the handling of requests in a thread pool. This lets you measure the (average) duration of handling requests (which includes keeping track of the response time of *downstream* services), and also lets you quickly reject incoming requests (e.g. returning HTTP 503), whenever the average request-handling-duration already exceeds your predetermined SLAs, or whenever there are already too many queued requests.
    - Technique #3 (my idea, not from the book): for Kubernetes (or other systems that distinguish between readiness (send-me-traffic) health checks and liveliness health checks): have "cascading readiness health checks". Example: You have a call chain that consists of the reverse proxy (Ingress), service A and service B. If every functionality of A also requires B, and the readiness probe of B is already indicating problems, then the readiness probe (*not* the *liveness* probe) of A should also indicate the problem, even if A's other dependencies (e.g. its own database) work fine.
- Service discovery (P. 188):
    - Generally, don't attempt to write your own SD. Use existing implementations. These might come integrated with your orchestration platform (e.g. Docker Swarm or Kubernetes), or use services such as ZooKeeper/etcd (which, considering the CAP theorem, are pessimistic CP systems), or e.g. Hashicorp's Consul (AP system)

10. Control plane

- "Control plane" = all the software and services that run in the background to make production systems handle load successfully. (P. 193)
- You should also have (production-level) SLAs for development-related tools/services/environments. If these go down, developers can no longer do their job, causing major problems. (P. 200)
- Observability (P. 200+): the book makes several notes about observability (called "System-wide transparency"), but with much less details than dedicated books (such as the *Practical Monitoring* book, covered here).
    - They recommend focusing on the business (monitor revenue) and what your users actually experience. Divide your business process into stages, making sure that there are no rapid declines from one stage to the next. (P. 202).
    - Each group of people in your organization has different needs (e.g. accounting and marketing may need different kinds of alerts and dashboards).
    - Metrics: it's impossible to create metrics for *everything*, but there are a few heuristics to decide which metrics to expose (P. 205):
        - Traffic indicators: Page requests, page requests total, transaction counts, concurrent sessions

        - Business transaction, for each type: Number processed, number aborted, dollar value, transaction aging, conversion rate, completion rate
        - Users: Demographics or classification, technographics, percentage of users who are registered, number of users, usage patterns, errors encountered, successful logins, unsuccessful logins
        - Resource pool health: Enabled state, total resources (as applied to connection pools, worker thread pools, and any other resource pools), resources checked out, highwater mark, number of resources created, number of resources destroyed, number of times checked out, number of threads blocked waiting for a resource, number of times a thread has blocked waiting
        - Database connection health: Number of SQLExceptions thrown, number of queries, average response time to queries
        - Data consumption: Number of entities or rows present, footprint in memory and on disk
        - Integration point health: State of circuit breaker, number of timeouts, number of requests, average response time, number of good responses, number of network errors, number of protocol errors, number of application errors, actual IP address of the remote endpoint, current number of concurrent requests, concurrent request high-water mark
        - Cache health: Items in cache, memory used by cache, cache hit rate, items flushed by garbage collector, configured upper limit, time spent creating items
- Make use of automated (repeatable) builds, and automated (canary) deployments (P. 208/209)
- Command & control APIs (P. 210): only applies to those services that start slowly (or that need a long warm-up period before they have good latency, e.g. JVM-based services). Here, *restarting* the service instances on each configuration change would be hurting the overall system performance. Instead, it's better to add a C&C interface to those services, e.g. implemented as HTTP REST API, to which you send control commands.
    - Examples for commands are:
        - Reset circuit breakers
        - Adjust connection pool sizes and timeouts
        - Disable calling specific external services
        - Reload / change configuration
        - Start or stop accepting load
        - Set or disable feature toggles
    - If you have *many* service instances, rather than having scripts which individually send commands (point-to-point) to each instance, think about using a *command queue* and a pub-sub messaging approach. However, be careful to avoid the *dogpile* anti-pattern.
- Consider using existing platforms (such as Kubernetes or Docker Swarm) and fully embrace their ecosystem and their native way of thinking.

11. Security

- Security is something that needs to be baked into your software and SDLC from the start - it cannot be added at the end. (P. 215)
    - Also, you need to *continuously* work on security.
- The book rehashes the OWASP Top 10 - since it is already outdated, it makes more sense to look up the OWASP Top 10 directly on its website (skipped summarizing it here).

12. Case study

- Author tells a horror story where deployment was done completely manual, involving around 20+ people who have to contribute to getting the deployment out. That would cost about 100k USD per deployment.

13. Design for deployment

- Avoid "planned downtime" that is caused when you have to temporarily shut down the system to deploy a new version. To your end-users, downtime is always bad, no matter whether it is planned or not. (P. 242)
- The gist of this chapter is: as the *development* team should treat (zero-downtime) *deployment* aspects as a feature of the software, not as something that only the ops team is concerned with. That means that the dev team needs to take part in creating the CI/CD pipeline and all its tasks and tools.
- The book generally recommends the "immutable infrastructure" approach over the mutable server/infra approach, because it allows for better reproducibility. (P. 246)
- *Continuous Deployment* is important because it reduces the time between <developer commits code> and <code runs in production> to a minimum. This is good because undeployed code is unfinished inventory, a liability with unknown bugs and unknown behavior in production. It might even be a great implementation of a feature nobody wants. Until it is deployed, there is a high degree of uncertainty. Also, big (rare) deployments are always riskier than many smaller deployments. (P. 246)
- You can decide whether your CD pipeline *automatically* deploys to *production*, or whether a manual approval is necessary. It depends on your organizational context. Manual approvals slow the deployment process down, but if the costs of breaking production is (by your estimation) much larger than the costs of moving slower, you should have approvals. (P. 247)
- A deployment is often a lengthy process (it may take many minutes). Not only do you have to roll out many services (in parallel) that make up your system and do some additional preparation or clean work (macro-level), but you also have to consider the 4 deployment phases of each individual service (micro-level), which are executed *sequentially*. (P. 248/249):
    - 1) Preparation: for mutable infrastructure: copying files into place. For immutable infrastructure: loading the new VM/container images to the hosts where they are

needed.

    - 2) Draining: drain traffic from old instances, usually is very quick with stateless microservices, but for old Java monolith with stateful sessions it can take a long time (you should choose an upper limit)
    - 3) Update: apply the changes. For mutable infra, this could e.g. be moving files or bending symlinks, for immutable infra this phase does not really exist.
    - 4) Startup: start the new release, which often takes some time until it is ready to handle incoming traffic
- DB schema evolutions must be handled very carefully.
    - For *relational* DBs (tables), make sure that you don't rename or delete tables or columns, but instead only do additive operations, followed by cleaning outdated data much later (e.g. several days/weeks later, once you are sure the destructive operation is safe). Instead of deleting something, simply don't use that column / table. Instead of updating/moving data, create new columns/tables and *copy* the data there. To ensure that data written by old code is also transformed to the new structures (and vice versa), think about using DB *triggers*. Be careful to avoid *cyclic* effects of these triggers. (P. 250)
    - For *NoSQL* "schemaless" databases (which are not really without a schema - instead of the DB enforcing a schema at *write*-time, your app code enforces it at *read*-time), you have options such as (P. 252-255):
        - Add backwards compatibility support to your code, leaving the old objects as they are in the DB, transforming them to the new versions only in memory. The downside of this approach is that you have to write a lot of code, and write tests for every entity version. Also, processing of objects with an older schema takes additional processing time.
        - Let your newly deployed code only handle objects with the new schema, and write a migration routine that migrates all objects in your DB at once, some time during the deployment. The main problem is that this could take minutes or even hours, and cause down-time
        - Let your newly deployed code handle both the old and the new schema. You still need to look out for consistency problems that could happen if you let the migration and the deployment procedure run concurrently: there is unpredictable behavior when the migration takes long, and also the (rolling) update to new code takes long (and you need zero-downtime). For instance, old code might read objects which have already been migrated, having the new schema (which it cannot handle). Therefore it is usually recommended to separate the "migration procedure" from "code deployment", timing-wise, doing the code deployment *before* the schema migration.
            - You can choose between running a migration *batch* job in the background, or do an *incremental* approach (called "trickle, then batch" in the book). Here the new code migrates old-schema-objects that it comes in touch with *on-the-fly* to the new schema. After the new deployment is in production for a while, you additionally apply a batch job that migrates all other objects still

        having the old schema.
    - Both the incremental and the batch approach of course also work for *relational* DBs.
  - Always make sure to test schema migrations on production(-like) datasets. The problem with production data is that it may break invariants/assumptions, that you either a) have never considered/checked, or b) are only checked in your code *now*, but were not yet checked years ago when the data was created. (P. 252)
- Handling the serving of static (web) assets (e.g. HTML, CSS or JS files) for rolling-update deployments: P. 256)
  - These frontend-assets are often tightly coupled to e.g. frontend services (that generated the HTML that contains links to the frontend-assets), backend services, or there could be problems with the load balancer.
  - "Cache busting" is a commonly-used technique to ensure that clients load the correct (updated) version of a web asset, e.g. building a reference such as <link rel="stylesheet" href="/styles/app.css?*v=a5019c6f*"/>
    - A clever trick can be to use the short Git SHA as version identifier.
  - If your reverse proxy is unaware of rolling updates, it could happen that it directs the request #1 of a user to the new service instance (that delivers HTML that references new web assets), and requests #2 to #n (that load these assets) to older service instances (that don't have these new assets yet). There are several methods against this:
    - Configure *session affinity*, where the reverse proxy transparently adds cookies or other headers to the requests and responses, to be able to correlate a specific user session, and always send it to the same frontend or backend service.
    - Ensure that the new assets are always deployed to *all* hosts/sites before rolling out new code referencing these new assets.
- Clean-up phase (P. 259/260)
  - After a rolling update/deployment has completed, you will usually want a clean-up phase, e.g. some hours/days later, after you are sure that you won't need to roll back to an older version (because the new feature works). This involves aspects such as:
    - Finalizing the DB schema, e.g. deleting no-longer-needed tables, columns, views, aliases, triggers, or foreign-key constraints.
    - Delete no-longer-needed code that was able to handle outdated-schema entities from the DB (particularly when you use NoSQL DBs)
    - Review feature toggles, delete those you no longer need, because the feature is now always enabled anyway.

14. Handling versions

- This chapter is mainly about breaking vs. non-breaking API changes.
- There are many examples of breaking changes (P. 265):

    - Suddenly rejecting a technical way of doing a request, that used to work (e.g. no longer accepting gRPC calls, or no longer accepting XML encoding in a REST call)
    - Changing request URLs (for HTTP-based APIs)
    - Adding *required* fields to requests that used to be optional or non-existent
    - Forbidding *optional* request data that was previously allowed
    - Removing fields in the response that used to be guaranteed
    - Requiring an increased level of authorization
- The opposite, *non*-breaking changes, are achieved by generally accepting more input data than before, or to return more data than before.
- As soon as the *implementation* of a service is up, its users take the way the implementation works for granted, treating it as the "de facto *specification*". Even if there is an official *specification*, your service users will rely on implementation-specific behavior. Therefore, even if you detect a mismatch between your implementation and the specification, you should avoid silently "fixing" the implementation with changes that would normally be considered *breaking*. Instead, update the incorrect specification to reflect the implementation's actual behavior, then issue a new/fixed specification (with a matching implementation), that contains the breaking changes. (P. 266)
- Make sure to specify your service API with precise modeling languages, e.g. OpenAPI for REST APIs. (P. 266)
- Have dedicated testers (who were not massively involved in the service's *implementation*) write contract tests for your API, to minimize the chance that there are mismatches between your API's specification and implementation. (P. 266/267)
    - Do this for both your own APIs, but also write such contract tests for *third party* APIs that you consume. Regularly run those *third party* API contract tests, to automatically detect once their APIs start behaving differently than expected.
- For HTTP-based APIs, there are several technical approaches to versioning your API. The most common (and recommended) one is to put the version into the URL. There are other, header-based alternatives, e.g. using the *Accept* Header for GET (e.g. "application/vnd.myapp.myapi.v1"), and *Content-Type* for PUT/POST (to indicate the API version you are sending the serialized objects to that are contained in the request body). (P. 269)
- Some tips for introducing breaking changes (P. 270):
    - If the client does not specify a desired version, assume a default, e.g. the oldest version that is not yet deprecated.
    - Keep supporting both the new and the old versions for a while.
    - For your own APIs, write tests that mix calls to the old and the new API version on the same entities. A common problem is that entities created with the new API endpoints cause problems when trying to access them with old API endpoints.
    - Not covered by the book, but consider the tips presented in https://blog.stoplight.io/deprecating-api-endpoints:
        - OpenAPI supports the "deprecated" keyword, for endpoints
        - "Sunset" header, indicating the date once the API stops working
        - "Deprecation" header which can either be true, or indicate the date (usually some datetime in the past) when the API has become

deprecated. It’s a good idea to then also set the “Link” header that contains the link to the updated API endpoint.

15. Case study

- Authors tell a story about a failed launch where servers crashed when there were too many concurrent sessions.
- Real-world learnings about *load testing*: (P. 281+)
    - Typically, business people will tell you a ballpark for the number of concurrent users. There are several problems:
        - Their estimation may be quite wrong (as in: too low) → you should strive to ensure that your software can take a multiple of the estimated load.
        - It’s better to use a different metric, e.g. *requests per second*, than *concurrent users*, because you cannot properly measure the number of concurrent users (HTTP is stateless, users don’t explicitly mark the end of their session, thus you can only estimate it by using session cookies and applying timeouts for declaring the end of a session - but this is an *overestimation* - you will “measure” more sessions than there are *actual* concurrent users).
    - Load testing is both an art and a science. You can never fully replicate *real* production traffic - there are always unexpected things that happen in production. All you can do is:
        - **Traffic analysis** (from past production and test environments): you learn about typical page-flows / access patterns, number of requests per session, conversion rates, pause-time-distributions (where customers think about what to click next).
        - **Experience and intuition:** helps you extract mental models, choosing which aspects of the above *traffic analysis* are actually important to load-test, and how to tune your traffic model to simulate “worst case” scenarios. Here you would also estimate how many of the users go all the way through checkout (checkout is the most expensive phase, because it involves the most page views, and the most (long-lasting) interactions with external services).
    - When simulating, do not only build load test scripts that mimic real users with well-behaved browsers. But also simulate other actors, such as scrapers (good ones like search engines, and unwanted ones like third party bots that scrape your page for some reason, e.g. a shop bot that gathers the lowest prices for specific products). Expect that such other actors do not use real, well-behaved browsers, but may e.g. not respect a session cookie, causing a new session to be created on your server on each request.

16. Adaptation

- As the business in the real world changes constantly, so must our software, for it to stay useful. This applies actually not just to the software itself and its design, but also to the involved people, processes, and tools. (P. 289)

- Accelerating the *decision cycle* (P. 290+):
    - Most companies have some variant of a decision cycle, e.g. the Plan-Do-Check-Act (PDCA), or Observe-Orient-Decide-Act (OODA).
    - Various processes and tools help shorten different phases of the decision cycle, e.g. DevOps, or lean and agile development methodologies.
    - Optimizing the "check" phase (where you verify your hypothesis) is often overlooked: you need to avoid moving fast while being clueless at the same time, which would happen if you do not get *good* feedback from the customer (or not getting it *fast* enough). But you need that feedback, to plan your next iteration. If you find that your development and deployment iteration speeds are faster than you can collect feedback, spend time on e.g. an experimentation platform to speed up the observation and decision-making process. (The book unfortunately does not get any more specific than that). (P. 292)
    - Larger companies can benefit from a *platform team* (P. 293). It treats the various development teams as customers, providing them a platform as a service, which lets the developers build/deploy/operate their code more easily. The platform team builds APIs (of the platform) and maybe (CLI) tools (e.g. for provisioning) for the dev teams. The platform team does only build the platform, not the application itself, and is also not in charge of the application's availability.
        - The feature set of such a platform includes:
            - Allow easy specification of compute capacity (e.g. high RAM vs. high I/O vs. high CPU/GPU), or storage (e.g. blob vs. block vs. file systems)
            - Workload management, autoscaling, placement of service instances, overlay networking.
            - Observability, including metrics/logs/traces collection, indexing, search and visualization.
            - Message queueing
            - Traffic management
            - Network security
            - Service discovery (e.g. DNS-based)
            - Other common service gateways, e.g. for sending emails or SMS
            - IAM, including users, groups, RBAC
        - Note: you can also buy platforms-as-a-service products, but this does not make your own platform *team* obsolete, it only gives it a head start.
    - Use incremental rollout / deployment strategies which reduce the impact of bad releases (which were not prevented by your automated tests). Concrete implementation strategies are *canary deployments*, or *blue/green deployments* - both have in common that you are incrementally directing traffic to more and more of the updated service instances, aborting the rollout in case you observe too many errors. (P. 295)
    - Consider sunsetting services instead of trying to fix them: sometimes it's better to shut down specific parts of your application that have shown not to work well,

rather than spending more money to fix it. If you have broken down your application into *many* smaller services, you can make the "cuts" in a more fine-granular way. (P. 296)
    - The main point about the famous “2 pizza teams” (from Jeff Bezos) is often misunderstood as “having *small* teams” (to avoid communication overhead). But actually, such teams only work efficiently if they can act *completely autonomously* (all the way to production), not having to wait for others (e.g. for an admin setting up a database). Having a *platform team* (see above) that has built a *self-service* platform is a great enabler for such teams. (P. 299)
    - Efficiency has the negative side effect that the resulting process becomes non-generalizable and harder to change. A concrete example are build pipelines, which make you efficient (especially the optimized pipelines) but they are tailored to a specific VCS, CI/CD platform and deployment platform, being not portable at all. Thus, when choosing the concrete implementation technologies that make your process more efficient, check within your company and with your customers if your choices are really good, for the next few years to come (e.g. whether to tie yourself to AWS or some other provider). (P. 300/301)
- Building a *system architecture* that can easily be adapted over time:
    - Sidenote: rather than thinking along the lines of “form follows *function*”, in practice you often have “form follows *failure*”. In practice, very often the final architecture/design is one that results from having fixed all the flaws of previous iterations, rather than being optimized to achieve its desired function in a particularly great way.
    - Take a look at various architectural styles to find the one that best suits your requirements, e.g. not only monoliths, but also microservices, microkernels, or event-based architectures. (P. 303)
    - There are ways of *modularizing* your application:
        - Check out https://scs-architecture.org/ for self-contained systems, where the basic idea is to have a breakdown of your system into multiple completely independent systems, along the *functionality* axis. It’s similar to microservices, but there are less SCS modules than there would be microservices, and there is also less communication between the SCS modules (compared to microservices).
        - The *Design Rules* book by Baldwin/Clark is discussed, which identifies many “module operators”. Examples are: (P. 308+)
            - Splitting of modules: break a module into several sub-modules. Replace the former module with a *facade* whose interface is the same as the module had (before splitting it). The facade simply delegates the work to its internal sub-modules. This improves *failure isolation*: before, if the module was broken, the entire feature did not work. After the splitting, we can limit the failure to a subset of the features.
            - Module substitution: replace one module with another one, assuming that they both implement the same interface. There could be various reasons for why you substitute the module, e.g.

        because the replacement is cheaper, better, etc.
        - Choose a good *axis* when breaking down your system into modules, such that the division facilitates adding / replacing / removing modules. A typical example would be to choose *functionality* as axis (as done by the *SCS architecture* approach, see above). A counter-example would be to decompose your system along *technical* lines, e.g. having a database module, a HTML-rendering-module, a logging module, etc.
        - Extract common functionality that is implemented (as slightly varying copies) in different submodules. As example, consider that many of your modules internally implement an A/B-testing mechanism. It makes sense to extract this to an “Experimentation service”, that all your modules now simply plug into.
- Build a suitable *information architecture* where you decide which data structures to create. (P. 313)
    - This section is written in a confusing way. It touches on event-based architectures (and that you have to pay attention w.r.t. the versioning of the payload of events). But the *Designing Data-intensive Applications* book does a much better job at explaining these things.
    - The author describes well-known concepts (without mentioning them) in convoluted ways, such as HATEOAS for REST APIs, or *bounded contexts* from Domain-Driven-Design.

17. Chaos engineering

- Chaos engineering is about injecting different kinds of chaos into the *production* system, to verify that the system (as a whole) stays operational, and if it does not, to learn how the system breaks. Many problems only show up in the whole system (they cannot be observed from individual components, or be induced by unit or integration (stress) tests). (P. 326)
- Prominent modes of chaos are: (P. 328)
    - Chaos monkey: kills instances of (auto-scaled) services
        - You do this because you expect a certain set of instances to fail anyway, for various reasons, such as bugs, or because the cloud’s hypervisor simply kills it for unknown reasons.
    - Latency monkey: injects latency into your internal network calls. Discovers problems such as:
        - Your service not properly using its “fallback” behavior when its calls to other services timed out
        - Problems caused when your service does several calls in parallel, and now the results arrive out of order.
    - Janitor Monkey (not further elaborated in the book, but it seems to be a service that auto-cleans unused resources. A set of rules define what counts as “unused”, and deletion does not happen immediately, but it is scheduled,

    notifying the owner of the resource ahead of time.
  - Conformity Monkey (not further elaborated in the book, seems to be a validator that verifies that cloud resources are properly configured)
  - Chaos Kong (not further elaborated in the book, but it seems to be like a Chaos *monkey* that kills an entire region of your cloud provider)
- Opt-in vs. Opt-Out (P. 329)
  - When your organization is new to chaos engineering, make it opt-in, as this has a lower level of resistance. With enough success stories, you can make it opt-out. Once Opt-out, every service in production is subject to e.g. a Chaos Monkey. There is a way out (the opt-out), but those services are “stigmatized”, end up on a list in a database, which engineering management should regularly review, reminding the service owners of opting back in.
- Adopting chaos engineering has 4 phases (P. 330+)
  - Prerequisites:
    - Consider *not* using chaos engineering at all, if you are in an environment where (almost) every single request has a very high business value. In other words: if breaking the system also breaks your bank, don’t do it.
    - Limit the exposure (“blast radius”) of the chaos tests.
    - Ensure that you have really good monitoring. Be careful that the injected chaos does *not* affect monitoring itself. -> monitor your monitoring during your test sessions (during testing, it would be very suspicious if all your dashboards remained completely “green”).
    - Have a solid recovery plan. The system might not recover back to a healthy state automatically, after ending the test session. You must know what to restart manually (or other clean-up procedures).
  - Design the experiment:
    - Build a hypothesis, e.g. “replicated services should not be negatively affected by instance failures”, or “the application remains responsive, even under high-latency conditions”. Focus on the *externally*-visible behavior, not the internals.
    - Think about the evidence that would cause you to reject the hypothesis.
  - Inject the chaos (run the experiment):
    - A hard problem is to make the concrete choices as for which specific service instances, or network calls are interesting enough to inject a fault.
    - A simple approach is to use *randomness*: randomly select services, connections, etc. But randomness alone is not sufficient, you also need more *targeted* fault injections. A computer or algorithm cannot help you here, you need human thinking: analyze your system, e.g. looking at the call trees, or collecting actual traces in practice.
  - Automate and repeat:
    - If a chaos test uncovered a problem:
      - Implement a fix
      - Check whether other parts of your system might have the same kind of problem.
      - Ensure that a similar test finds that the problem is really gone.

- You can extend chaos engineering to the *humans* in your organization, too (P. 335).
    - E.g. randomly select 50% of your people and tell them that they shall not respond to communication attempts.
    - You will quickly discover processes that no longer work once specific people are out of office. Try to fix these processes, e.g. by improving documentation, or changing roles, or even automating processes that used to be manual.
    - You should still have ways of aborting the chaos test, e.g. via a “code word”.