A GUIDE TO SOFTWARE ARCHITECTURE

TABLE OF CONTENTS

WHAT IS ARCHITECTURE?

Seems like a natural place to start. When one thinks of software architecture and what a Software Architect does, they usually think of diagrams, oftentimes a very complex tapestry of lines, boxes and various colors as the Architect stands back proudly admiring the beautiful art they created. This is exactly what this guide aims to end. By bringing structure, rigor and shared understanding into our craft, we all benefit. Let's get into it.

First things first: Your system has an architecture whether you consciously plan for it or not. It is an emergent property of the system and emerges from decisions made explicitly or arbitrarily. Your architecture is what will enable or hinder your ability to deliver on business commitments and will likely influence how developers feel about developing in the system. Yes, developers have feelings.

I will assume your business desires an architecture which enables movement unimpeded by technology and simultaneously is a delight to develop in. That is precisely what a Software Architect should provide. Through an understanding of the business needs, technology realities, and various constraints, they discern the desired architecture and the steps required to get there. All decisions should have traceability to a business case. Technology, and hence software architecture, is just a tool that exists to enable the business to deliver customer value. Each architectural decision must be made through that lens.

My background is in pure mathematics and so I tend to attack most problems from that angle. This may or may not be useful to you, but I include it as an example. As a Software Architect, I view the design problem as an equation: S(*x1, x2, …, xn*) = T(*y1, y2, …, ym*) where S computes the strategy and T computes the agreed upon target state, of which there are infinitely many solutions.

It is the Software Architect's responsibility to determine, not the functions, but the variables and the trade-offs of values which can be made to satisfy the equation. This is achieved through interviews, investigations, conversations, critical thought and battle-scarred experience. From those values, a strategy and a target state *emerge*. With the context set, I present this guide to software architecture that I have honed from various sources much smarter and more talented than myself over the past decade. It contains the strategies and documentation I think are necessary to accurately design, document and articulate a software architecture. Not everything will always apply, and your design may need to expand in some areas. Knowing which knob to turn and by how much is key but that comes with experience. No book, article, or video can teach that. You must practice.

SYSTEM OVERVIEW

This is where every architectural design must begin and an area that is revisited often to validate decision making. If you are starting off a design talking about AWS, Kubernetes, gRPC, etc. **STOP IMMEDIATELY**. Those decisions come later and in nearly all cases are inconsequential if you do not understand the business problem and the customer value that is desired from the system.

Do not skip this. Do not skip this. Do not skip this. There's three warnings.

BUSINESS CASE

Who should provide this? Business Analyst, Product Manager/Owner, or equivalent

Who should document this? Software Architect

What types of questions should this answer?

- What customer value will this provide?
- What business advantage will this provide?
- Is this a catch-up (competitor) or innovation play?
- What type of analysis or research was conducted?
- Who really wants this?

Tips: Ask questions when you feel there are gaps in research or justification but don't be condescending or a blocker. Your goal here is to understand and document the **why**. Don't act like things would change if the Product Manager handed you a 50-page user research report—it's not like you would do more than skim it anyway. This is where experience comes in. Try to be a resource that is sought out, not a required approver.

STAKEHOLDERS

Who should provide this? Business Analyst, Product Manager/Owner, or equivalent

Who should document this? Software Architect

What should this contain? All stakeholders (people come and go, roles are fairly constant) interested or affected by the proposed system. For example: CEO, CPO, Director of Product, Director of Engineering, Sales Managers, Customer Success Managers, Marketing Managers, etc.

You may not know who all the stakeholders are in advance. This should be revisited and updated as new information is learned.

KEY PERFORMANCE INDICATORS AND SUCCESS MEASURES

Who should provide this? Business Analyst, Product Manager/Owner, or equivalent

Who should document this? Software Architect

What should this contain? A clear definition of success. What are we seeking to improve by introducing this system? Revenue? Customer satisfaction? Efficiency? Etc.

INTERVIEWS

Who should provide this? Software Architect

Who should document this? Software Architect

What should this contain? Questions, concerns and requirements from all stakeholders. What you document is up to you but what I find useful is focusing about 75% on goals and 25% on functionality. If your interviews end up being nothing but a user story backlog, that is a huge red flag and is a sign you need to improve your interviewing skills to extract the essence of their concerns.

Understand what your stakeholder wants to achieve, not how they think it should be achieved. However, including a diverse set of ideas of functionality can lead to a more resilient and extensible architecture so hear them out and keep an open mind.

If you can go back and understand the notes taken in the interview, there's no wrong way to document it, but here's what I use when conducting interviews:

System Name	**[SYSTEM NAME]**
Stakeholder	[NAME], [TITLE], [ROLE]
Drivers	[what drives their interest in this system? Does it accelerate their roadmap? Does it reduce technical debt? Etc.]
Concerns	[open-ended; extract from discussion of intended usage. E.g. must remain available during traffic spike patterns]
Desired functionality	[stack-ranked desired features]
Relevant KPIs	[LIST OF KPIs]

ARCHITECTURE PRINCIPLES

These should be set at the Enterprise or Organization level. It would be very uncommon, but not impossible, to have a set of architecture principles for every system. Once they're written they can just be linked or referenced within your design documentation.

HOW TO USE

Who should provide this? It depends on your organization. Usually a Chief/Enterprise Architect or an Architecture Review Board (ARB). But in smaller organizations, it may be you.

What should this contain?

- Engineering teams use architecture principles to guide their decisions concerning design
- They need to be at a level low enough that aids decision making but high enough to have broad applicability
- No more than 10, aim for 5.
- Each principle should contain: Description, Rationale and Implications

PRINCIPLES

Include your principles here. Here are a few examples I've used at multiple organizations:

1. **Choose Single-Responsibility Systems**
    a. **Description**: Similar to the single-responsibility principle for objects, this system-level design principle encourages creating or acquiring loosely coupled systems that interact with other systems through standard protocols
    b. **Rationale**: Systems must be upgraded or replaced as business needs change. Systems that perform multiple business functions are harder to upgrade and replace.
    c. **Implications**: Systems utilize standards for intersystem communication. Systems can be replaced with minimal disruption to other systems.
2. **Design for Emergent Reuse**
    a. **Description**: Emergent reuse is the ability to identify existing systems that can be used in implementing new systems. Utilizing existing systems in a new application is more effective than designing a new application
    b. **Rationale**: Reusing capabilities in different areas reduces the overall system maintenance cost.
    c. **Implications**: Systems that use open-standard interfaces are easier to utilize and adapt
3. **Demand-Driven Releases**
    a. **Description**: Software releases must be in tune with the business demand. Technology or process should not inhibit the business' ability to deliver on commitments
    b. **Rationale**: Systems must be adaptable to change so the business can react quickly to market and business changes
    c. **Implications**: Systems are flexible and amenable to change. Teams limit work in progress and work in smaller batch sizes following lean product development principles

DOMAIN DEFINITION

I'll spare the usual argument I make supporting Domain-Driven Design (DDD) as the supreme modeling and system decomposition methodology. You may choose whichever methodology you like so long as you accomplish the following:

- ✓ **Cross-functional shared understanding**. Product Owners, Software Engineers and each role in between should be able to give the elevator pitch of the system and describe its core capabilities. In other words, software modeling must be collaborative.
- ✓ **Immaterial changes in the business do not materially change the software model**. In other words, your software model reflects your business model. They may change together.
- ✓ **Build comradery and empathy within the team**. If your "team" is nothing more than an arbitrary grouping of individuals, you do not have a team. Software modeling is the ultimate team building experience. They must understand where the goal line is and how their activities and interactions with one another move the team closer to the goal line.

In my experience, no methodology ticks all three of those boxes as well as DDD. Going into the details of DDD is beyond the scope of this guide. **Use this book and have all Product Managers and Engineers read it.**

SUMMARY

The domain definition should be your first stab at constraining and defining the problem space. What is not included is oftentimes more important than what is included as it aids in keeping discussions and working sessions focused and productive.

Who should provide this? Software Architect leads this development, but contributions will vary according to stakeholder needs. In general, this is a shared activity among Architect, Product Manager and Engineers.

Tips: These session(s) will consist of many individuals—product leadership, engineers, and adjacent or interested department leads. Resist all temptation to discuss technology or implementation details. Apart from a few extreme cases where the problem is technical by nature, the problem space applies regardless of the solution and therefore discussing solutions provides no value to understanding the domain. Don't worry, there will be plenty of time to dig into the solution space once a clear vision of the problem space is articulated.

BOUNDED CONTEXTS

Your grouped concepts wherein each boundary, those concepts have their own ubiquitous language

CONTEXT MAPPING

Dependencies and relationships among your bounded contexts

EVENT STORMING

The natural events that emit from each bounded context and those interested in those events

UBIQUITOUS LANGUAGE

The unique way terms are used within each bounded context

HIGH-LEVEL SYSTEM ARCHITECTURE

We're finally ready to begin on the artifacts most think of when they hear "architecture". But we're coming into this process armed with:

1. A justified business case with clear measures of success
2. An identification of the relevant stakeholders and their concerns
3. An understanding of the problem space including what drives and motivates our target end user(s)

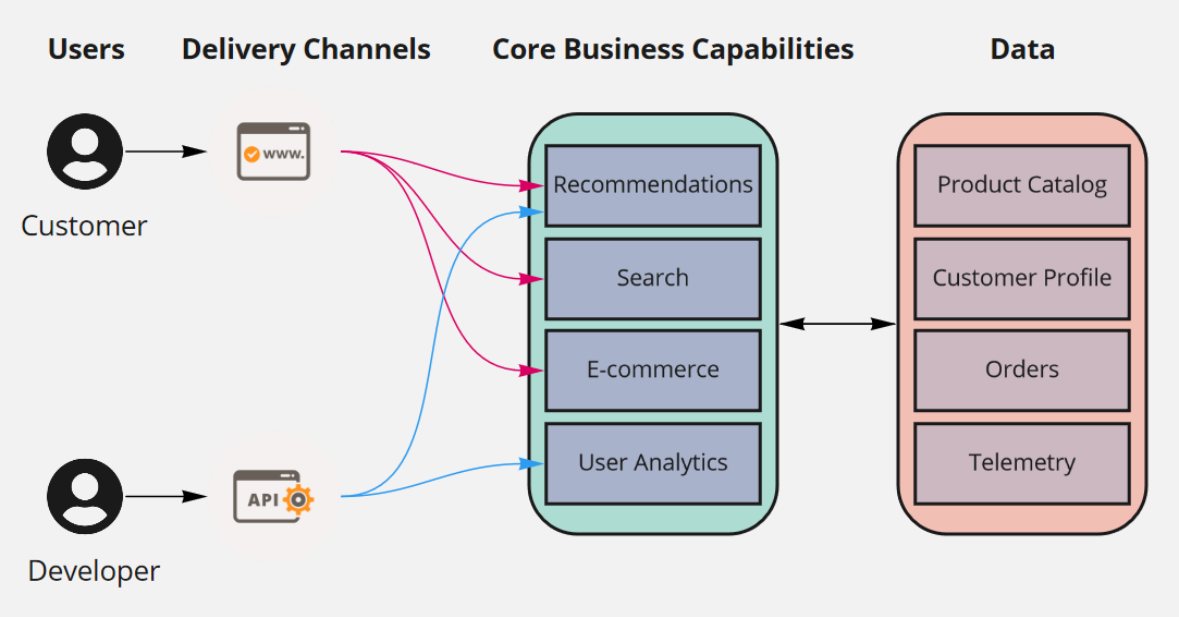
The above are necessary to design *the right* system. And *right* is meaningless without context. The above sets the context and later, will set the constraints which will ultimately guide our system and technology decisions.

Who should provide these? Software Architect

ENTERPRISE VIEW

The Enterprise View is a single view which gives the 30,000-foot view of the system. In general, it is comprised of the following elements: *Users/Personas, Delivery Channels, Core Business Capabilities, and Data*. Armed with this view and an understanding of the problem space, further detail of specific use-cases and interactions may be explored. Each element contained in the Enterprise View should be accompanied by a detail document which explains its purpose.

An example Enterprise View for a simple e-commerce application could look like this:



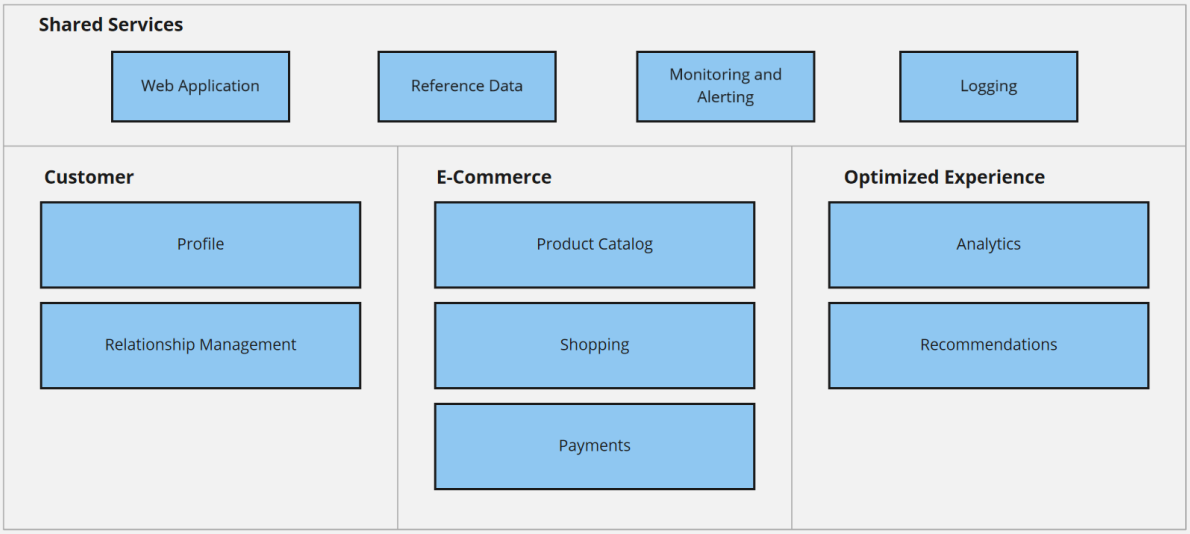
The Software Architect would then include a description of each element in this view providing the reader with a sufficiently high-level understanding of the system.

Tips: Full functionality of each element is unlikely to be known so don't spend too much time going into analysis paralysis at this level. Your goal here is to articulate the *essence* of the system to interested stakeholders. Business analysts, Product managers and Executives are likely to be the most interested in the Enterprise view but it can also be used in initial conversations with Engineering leaders.

COMPONENT DIAGRAM

The Component Diagram is a *static view* of the system to be developed and begins to expand on the capabilities documented in the Enterprise View. It still says nothing about how these components are implemented (e.g. as containerized services, shared libraries, or namespaced/packaged code in a monolithic application). However, it does begin to add structure and substance to what our system will ultimately look like.

The simple fictitious e-commerce system may have a component diagram that looks like:

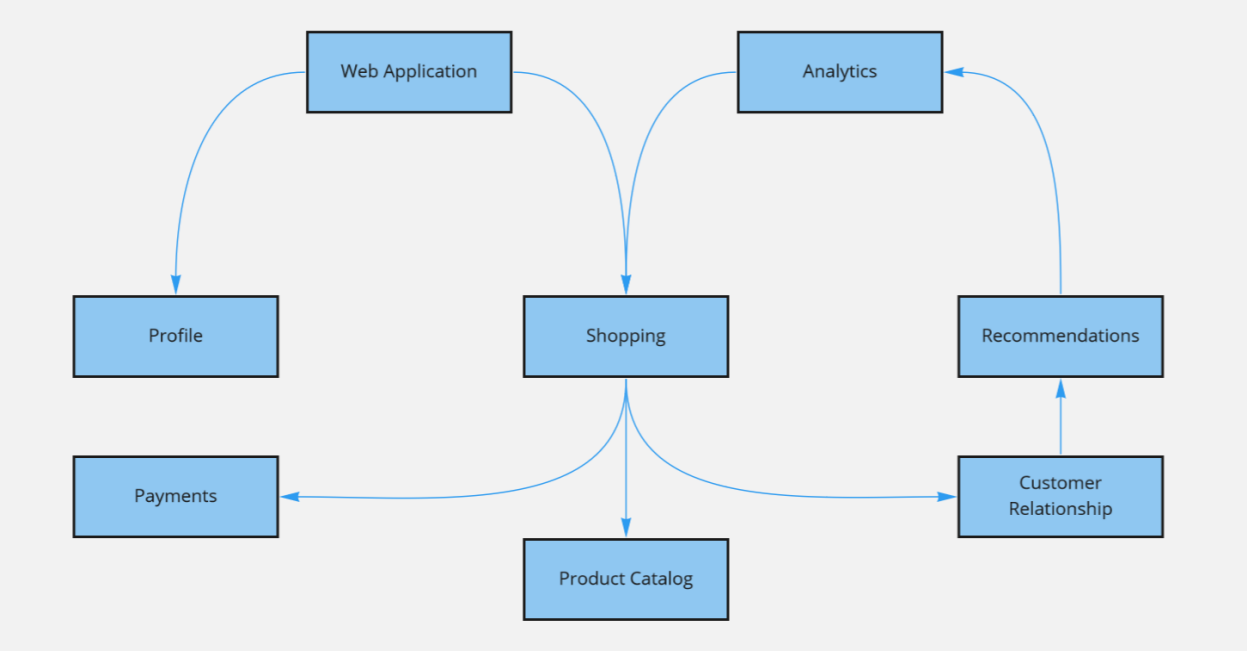


In this example, I've separated our components into three core contexts: *Customer, E-Commerce, and Optimized Experience*. These contexts would correspond with the Bounded Contexts identified and defined in the *Domain Definition* section through DDD Strategic Design activities. The components are formed by grouping the entities, aggregates and events which provide the functionality needed from those Bounded Contexts and identified and defined during DDD activities.

Just like the Enterprise View, this diagram must be accompanied by documentation for each component to include: Explicit responsibilities and intended usage such as via graphical UI, remote invocation, etc.

INTERACTION DIAGRAM

Building on the information provided by the Component diagram, the Interaction diagram shows how the components interact with one another as well as with (key) external systems. This is your typical “box and line” diagram but with boxes which have a clear definition and tie back to business KPIs.



Note here that this is still a logical diagram. It says nothing about the implementation. The directed edges from one component to the next simply indicate a dependency. Depending on implementation choices, that dependency could represent an in-process method call, a synchronous network request, a subscription to a stream, or (in most cases) an asynchronous handling of a message.

As with all our diagrams, this diagram must be accompanied by documentation for each connection (in and out). Some architects like to add more flavor to the above diagram by annotating each edge and labeling the data flow. I prefer to keep the diagrams as simple as possible and then document details in a text/tabular format which accompanies the diagram.

Tips: Keep an eye on the number of connections in your interaction diagram—both total and into each component. Each connection is a potential point of failure in your system which you will need to account for when you tackle quality attributes (non-functional requirements). The quality attributes will serve as the drivers to your architecture.

CORE USE-CASE DIAGRAMS

There's an art to extracting core use-cases because in general, people have a tendency to focus on details rather than the bigger picture. There's little value in enumerating every special edge case at this stage of the design. Such details are best done in an iterative agile fashion as new information is learned. Instead, we want to extract and dig into the macro use-cases which directly support and address the value proposition. By scoping our focus to this level, our design is immune to disruption from feature changes. The only way our core use-cases change is if the product or business fundamentally changes (the value proposition changes) and at that point, we should be looking at a new design anyway.

Even in a large system, you're rarely going to have more than 5 core use-cases. 3-5 core use-cases is the sweet spot. Once you've settled on the appropriate use-cases with your Product Management counterpart(s), it can be documented in a variety of ways. At this stage, it's best to focus on the tasks required to complete the use-case and document the relevant user concerns rather than to worry about how an activity will be implemented.

For example, in our simple e-commerce application, a core use case could look like the following:

UC-1: *Customer can successfully find, configure, and purchase a product*

Task: Search	Task: Product Selection	Task: Configure Variants	Task: Cart/Checkout	**Task: Place Order**
Description: Ability to search the product catalog and receive highly relevant results **Expectations:** Search should support phrase and synonym contextual matching along with exact product title/identifier matches. Results should be sortable, filterable and paginated. **Notes:** We will know in advance the fields which should be included in the search criteria and once entered, change infrequently with the exception of typos.	**Description:** Once the user is presented with search results, they're able to navigate to a product's details which will display information about the product--both static and dynamic information such as images, videos, description, reviews, similar products, etc. **Expectations:** At a minimum, the product detail needs to display a description and its possible variants such as color, size, etc.	**Description:** User is able to configure the product from its possible variants. **Expectations:** User receives feedback from the system regarding the variant configuration including but not limited to availability and price.	**Description:** User is able to store a configured product and quantity in a shopping cart to place an order now or in the future. **Expectations:** Cart functionality must be supplied without navigating away from the current product. When User is ready to checkout and place an order, they are navigated to a screen which prompts them to for their billing, shipping and payment information. **Notes:** Payment can be fulfilled using a variety of methods including but not limited to credit card, gift certificate, and loyalty points. Payments can also be split across multiple methods	**Description:** User receives feedback of order submission immediately. System routes order request to fulfillment and warehouse services **Expectations:** User receives notifications via their configured channels (SMS, email, FB messenger, etc.) of order status and any relevant communications from the warehouse or fulfillment services. **Notes:** Inability to fulfill an order is rare so the business would rather immediately take the order and deal with the off-chance an order can't be fulfilled on a one-off basis.

Who should provide this? Software Architect working alongside Product Manager/Owner

Tips: Resist all temptation to design around granular use-cases. The further down the granularity rabbit hole you travel, the tighter you couple your design to the *functional* requirements. This is a disaster waiting to happen as a change in the functional requirements (which happen frequently) are now finding their way into your design.

Your design should accommodate any variation or imagination of features and those features, should in some way, address a piece of the value proposition of the system. Therefore, if you structure your design around the value proposition, your design will accommodate any feature no matter how granular. This comes at the expense of the architecture "feeling" too high-level to some individuals.

If you encounter these individuals, you can include as an appendix a section which "checks your work". Here's what I do: Narrow down 2 or 3 of their primary concerns and show how one or more core use-cases satisfy their specific feature. This should build confidence in the process. I've seen this anxiety from individuals who have difficulty dealing with ambiguity or have experience building "architecture as a side-effect" systems common in startups or successful businesses that grew rapidly. It can be difficult to navigate these situations so be patient.

Using another mathematical analogy, you'll encounter individuals who want you to answer 1+4, 5+2, 3+8, 2+7, and 1+3. When you do that, "Oh, yeah! What about 9+22?!" Your goal is to answer $a+b$ for any integers a,b.

ALIGNMENT WITH BUSINESS CASE

While everything in the *High-level System Architecture* section has been aligned with, or more accurately, derived from the business case, this is your opportunity to make it explicit. The examples in this guide may make this section appear short but in a real system design, it will likely be a dozen or more pages.

I'm going to go out on a limb here and assert that your executive stakeholder(s) are not going to read the high-level architecture in detail. They may glance at the diagrams and skim some of the documentation but even that is rare. What they will care about though, is that the architecture you're proposing meets the business needs from a functional point of view.

Note: non-functional aspects of the system are *more important* to your software architecture, but you have mighty tools called "trade-offs" and "options". Those are covered later.

So, think of this as the executive summary of your system design. It's your one, two maximum, page mapping of business capabilities, value proposition and KPIs to logical components and core use-cases. Obvious questions which arise from this subsection should be answered in the preceding subsections.

Tips: Don't be that architect that tells an executive to read the documentation. Know your audience. A sufficiently detailed high-level system architecture is a great resource for Business Analysts, Product Management and Engineers but not Executives. A great way to build trust is to listen to needs and proactively meet those needs.

QUALITY ATTRIBUTES

"Quality is not an act; it is a habit" – Aristotle ~ 350 B.C.

This is the **most important** section for your software architecture and unfortunately, these areas are often treated as an afterthought during the software development lifecycle. I've always found this ironic because it's these quality attributes which drive, constrain and ultimately dictate your architecture. Quality attributes are the significant decisions which quantitatively and qualitatively dictate how well your system performs its tasks.

If you disagree with the assertion that quality attributes are what determine your architecture, consider the following proof: Assume your architecture is not determined by quality attributes. Then, if that were the case, no amount of refactoring could improve your system's ability to scale, change, or integrate with other systems. Anyone who has worked on a system of any significance will be able to confirm this is a blatant contradiction. Thus, proving the original claim. Convinced? Great!

So, can a "good enough" architecture exist without explicitly addressing these concerns as first class citizens? Of course. Recall that architecture is an emergent property of your system and seasoned engineers think about these issues almost subconsciously because unavailable systems (literally) wake them up at night.

That's not the point of this guide, though. I don't want you to get lucky and arrive at an appropriate architecture by accident or divine intervention. Instead, this guide intends on providing a checklist and systematic approach to creating an architecture that will stand the test of time. Technologies change but good design is timeless.

AVAILABILITY AND SCALABILITY

What good is a system if it cannot process requests made by your users? The ability to scale your operations with user demand is vital. However, determining the level of availability while juggling other factors such as cost, and complexity can be a difficult task.

For example, availability of 99% (3.65 days of downtime per year) is likely unacceptable for any system. But availability of 99.99% (52.6 minutes of downtime per year) could be prohibitively expensive and/or overkill for your system. Moreover, the definition of what availability means to your system needs to be defined (e.g. if I can purchase products but don't see recommended products or upsells at checkout).

You'll often hear about fault-tolerant systems. Some systems are fault-intolerant and that can be a strategy, too: if my application is in an inconsistent state, I simply restart it as no configuration is as clean as a fresh boot. However, most applications in practice have a desire to be fault-tolerant so that will be our focus.

So, what does fault-tolerance even mean? Nygard (author of Release It!) has a great discussion about it but I'll summarize: Faults are like cracks in a foundation. They're not ideal but they're not immediately user-facing. If enough faults occur, or faults occur at just the right spot, they can compound into behavior the user does see. We call these errors. Then, a fault-tolerant system is a system which can tolerate these faults and function as expected from the user's perspective.

With that in mind, what can we do to build fault-tolerant systems? It comes down to three primary areas: **Detection, Recovery** and **Prevention.** There are many battle-tested techniques for each area. Here are some examples:

- *Detection:* ping/echo, heartbeat, self-test

- *Recovery*: active redundancy, passive redundancy, exception handling, rollback
- *Prevention*: instance/node removal, exception prevention

This is a huge topic but the above should get you started and headed in the right direction. Luckily, many popular PaaS and container orchestration tools make detection and recovery relatively straight-forward and a matter of configuration.

Last note on the topic of availability, which again is a huge one: *what happens when your system fails to meet such an availability requirement?* There should be a penalty, but how do you define that contract and make good on your commitments? This is where the concepts of reliability and service-level-X come in: service-level indicators (SLIs), service-level objectives (SLOs) and service-level agreements (SLAs). This topic is beyond the scope of this guide but see the *External Resources* section at the end for some books that will give you all need to be successful.

INTEROPERABILITY

Most systems, especially SaaS systems, require a significant level of interoperability. At a minimum, you'll want your system to be able to leverage other systems (internal or external) to reduce the direct responsibility of your system. Eventually, the business could very likely desire that aspects of your system be exposed externally as companies transition from Apps to Platforms. Will you be ready?

This quality attribute is primarily an application-level attribute and can be achieved well using SOLID design principles. We'll cover this in more detail in the *Application Development Model* section but in particular you'll want to ensure your system provides well-segregated and well-documented clean interfaces that other systems can leverage.

Tips: Depending on your organization: you may be responsible for this design, you may only need to review this design, or neither. In smaller organizations, you will almost certainly be responsible for this design. In larger organizations, you may only need to review the design provided by engineering team(s). Or you may simply specify the expectation of service interfaces and engineering teams are completely accountable and responsible for the design. No matter the case, be sure to clearly express the intent and expectations exposed by your system at a meta-level. Standards for documentation, APIs, and service interfaces get you very near the finish line. We'll go into more detail on these topics in the *Development Practices* section.

MODIFIABILITY

You know those applications that engineers dread touching? Why? Lack of modifiability. Put another way, adding features, extensions, providers, etc. is an outright pain. This is typically the result of an application which grows organically without a plan. It usually backs a successful business and that makes sense, right? The business is successful, so they grow, the feature requests pile in, the system becomes more complex and so on until at a certain point everything grinds to a halt. Ironically, the most successful businesses have the nastiest architectures. It's not a problem until it is. And just like financial debt, there will come a day when it's time to pay up.

Modifiability is an application-level attribute much like the closely related interoperability attribute. If you follow Domain-Driven Design as this guide recommends, the components you design will lend themselves very well to modifiability.

Common tactics to achieve modifiability include splitting components up into smaller cohesive components and reducing coupling by depending on abstractions rather than concrete implementations.

PERFORMANCE

Today's users have no patience for an unresponsive or slow application. Remember the days of a webpage slowly loading for 10 or more seconds? Those days are gone. Full page load in less than a second or your user is checking their router. User waits more than 30 seconds for an email confirmation and they're resubmitting the form. Users of your system are going to be impatient so what are you going to do about it?

Performance is such an important quality attribute as it taps in to how a user *feels* about your application. Those feelings will certainly translate into reviews, NPS scores, etc. As the performance of your systems and applications have a direct business impact, this guide puts such an emphasis on understanding the user's pain points, motivation, and goals early in the design process. Working alongside a Product Manager, you will use that information to determine what performance means to your users for your core use-cases.

As the Software Architect, you must provide the response measures for performance. For example, you may have response measures for latency and feedback to the user: e.g., a particular operation must complete within 2 seconds, but optimistic feedback must be provided to the user within 100 milliseconds.

There's various tactics which can be used to achieve performance. On the backend, choices tend to fall within two categories: *controlling resource demand* and *managing resources*. This can be achieved by binding execution times (setting timeouts and limiting retries), introducing concurrency, data redundancy, scaling out compute instances, and so on.

Tips:

- Product Managers will rarely tell you the response measures, but they don't expect them to be unbounded either. It's your responsibility to begin with reasonable defaults for your industry and problem space. Further, it's best everyone is on the same page from the beginning so explicitly specify those response measures. Assumptions can lead to finger pointing when things go wrong. Clear and direct communication is a strong skill you should sharpen often.
- What was performant last month may be dog slow today. How do you get ahead of that? Application Performance Management (APM) is key. Set alerts for increases in trends in resource utilization and response times. The user may not notice a 99th percentile response time of 350ms over 290ms but it could be a signal to you. Best to be proactive.

ACCESSIBILITY

Accessibility is a quality attribute that gauges to what degree your application/system functions for users with disabilities or impairments. There are some baseline expectations, but your requirements are heavily dependent on your users and problem space—which again, is why this guide emphasizes understanding the user and problem space from the get-go.

With respect to web accessibility, the W3C has a thorough guide contained in the Web Accessibility Initiative (WAI). Mobile platforms have their own accessibility guidelines and features: see here for iOS and here for Android.

This falls more on the implementation side but as the Software Architect, it's your responsibility to ensure that these concerns are reflected in the application design in terms of abstraction and encapsulation.

USABILITY

Much like performance, this is less of an option and more of an expectation. Usability encompasses a lot of aspects, but the two pillars of usability which really make or break a user experience are:

- **Intuitiveness**: Does a user just "get it" when they use your application, or do you need to assign a customer success team member to train your user? Don't train your users, fix your application.
- **Feedback**: Some workflows are complex or long running and you must ensure your system accounts for that. Your user should never wonder what's going on or what to do next to achieve their desired outcome.

An application with an intuitive and consistent user experience along with appropriate feedback will give you a much better chance at providing an enjoyable user experience. From the get-go, I advocate leveraging user-centric design. It's a methodology and mindset which starts with the end in mind along with a significant understanding of the user needs.

Tips: It may not be immediately clear where your users are encountering usability issues. In general, the Product Manager/Owner will be the one working with the users, but you can help guide those conversations and outcomes by providing data.

Just like APM is key in detecting performance issues, client-side telemetry and analytics can help understand user behavior and uncover potential gaps in UX or functionality. I recommend the following guidelines:

- Make it easy for that data to be gathered and aggregated by use-case/workflow/desired outcome, user, user type, etc. Enable a query capability for Product Managers and Business Analysts
- Make it easy for A/B tests to be conducted

There are many tools which provide these capabilities out of the box. This guide doesn't recommend any specific service or product but will provide reference processes for technology evaluation (ATAM and CBAM).

DATA ARCHITECTURE

In almost all cases, data is the real product. This section treats data as an asset; setting teams up for success by providing the guidance or requirements for all scenarios which interact with your system's data ensuring its storage and usage is compliant, performant and cost-effective.

GENERAL PRINCIPLES

Just like the overarching *Architectural Principles* set in the beginning of this guide, this is your opportunity to set data architecture principles which apply to all components in your system. Engineering teams will use these as a guide to help them choose databases, design schemas and expose their data. The same intention and guidance apply to these principles as we discussed with the *Architectural Principles*.

Here's some examples to get you started:

1. **Components may not directly access other components' data stores**
    a. **Description:** There should be only one path for data to be mutated, stored, or queried. No backdoors. Components must provide well-documented interfaces which abstract data access.
    b. **Rationale:** Direct data access leads to highly coupled and fragile systems
    c. **Implications:** Components may evolve independently and store data in a format which is best suited for that component.
2. **Use a storage mechanism that fits the intended use of the data**
    a. **Description:** Teams may choose from approved database technologies the best database technology for their data
    b. **Rationale:** Favoring team autonomy and independence over strict homogeneity allows for faster time to market and performant data access
    c. **Implications:** A polyglot environment which requires a lightweight governance model for introducing new technologies

STORAGE, ARCHIVAL AND GOVERNANCE

The management of data plays an important role in nearly all systems. This section should be used to help guide teams with respect to management of their data.

The first concern of data management is the technology used for persistence, e.g. what database? This guide recommends you do the following:

- Provide a database technology option for each data requirement (ACID, BASE, Search, Cache, etc.)
- Provide a process for introducing a new type of datastore

Standards for data archival and governance likely already exist at least for certain classifications of data within your organization. If they don't, then you'll need to work with adjacent departments such as InfoSec and Risk to help develop these. Your responsibility lies in two areas:

- Ensure that the standards exist and are documented
- Help teams to classify their data (e.g. does a piece of data count as PII, fall under GDPR, etc.)

Tips: A simple decision tree for selection of one of your supported database technologies can be useful for teams.

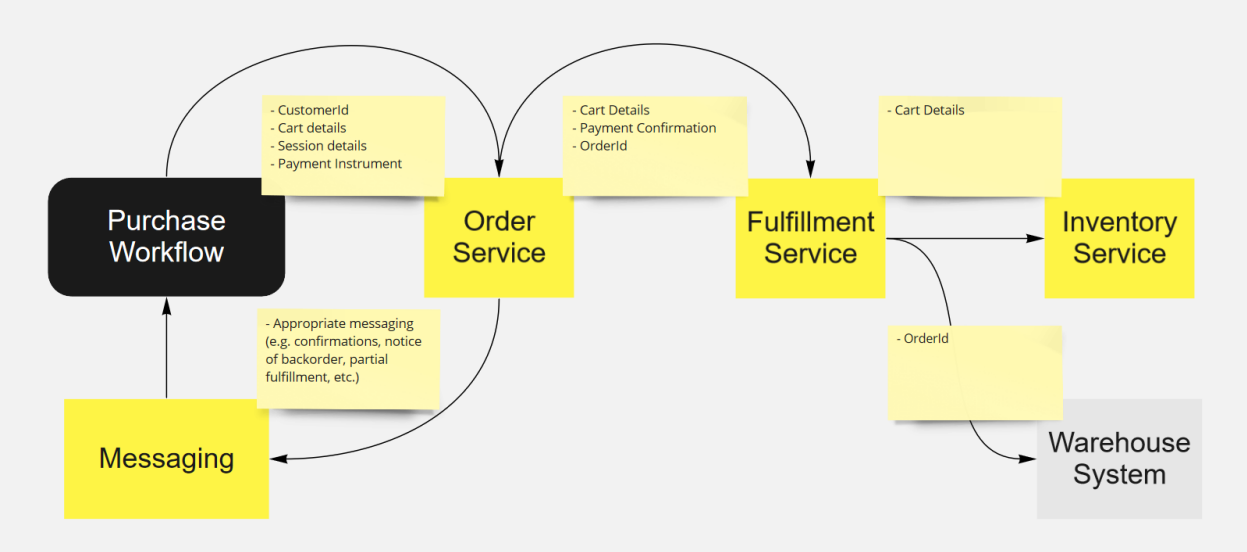
DATA MODELS

Much like this guide went over in the *Interoperability* section about service interfaces and API definitions, you may not be directly responsible for this. It's my opinion that anything beyond a logical data model is *too much* design for the Software Architect. That is, providing a schema to a team with a directive of "build around this" is certainly destined to fail. Such details are up to the engineering team.

However, if your team is comprised of all junior engineers, you will need to work with them to help guide their data modeling activities. This is where your experience and judgment come into play. You should provide your unique value required and then get out of the way. Do not be a roadblock, gatekeeper or ivory tower architect.

With that in mind, if needed or expected by your teams, an expanded UML interaction diagram which calls out core data flow could be helpful. Something like the following:

DF-1: Purchase workflow



SECURITY AND COMPLIANCE

Security and compliance practices are likely already set at the Enterprise level. In which case, simply referencing the applicable aspects is sufficient for this section. If you find yourself starting with a blank slate, use this section to guide application development. Engage with your CISO (or equivalent) as early as possible to fill any gaps. That role wields phenomenal power and establishing a trusting relationship with the CISO is key to ensuring that the system you design is implemented and shipped to customers.

COMPONENT-TO-COMPONENT

State the required security protocols and practices for internal communication among your application. Some examples to get you started:

- IP filtering
- TLS
- PGP
- SAML

You may also expand on the mechanism for achieving whichever practice you set. For example, using trusted certificates and/or obtaining a token from an authorization server which provides an allow/deny or scope access to dependent components.

Tips: Provide abstractions (libraries, sidecars, etc.) to enable this functionality for every application running in your environment. Failure to do so can have disastrous consequences if either by malice or naivety an unsecured or improperly configured application makes it to Production.

USER AUTHENTICATION AND AUTHORIZATION

Specify the mechanism(s) users must leverage to authenticate with your system as well as how components will authorize requests from users. A sufficient level detail, enough to be actionable, is required here. Some examples to get you started:

- Specify how Authentication is proved: Something you know (username/password), Something you have (two-factor/authentication key), Something you are (fingerprint, retina scan, etc.)
- The output of a proved Authentication: For example, a bearer token encoded as a JSON Web Token (JWT)
- How authentication is challenged: For example, signature validation of JWT
- How authorization is decided: Centrally, Distributed, etc.
- Types of authorization supported: For example, role-based or permissions-based
- The specific server or services used: For example, https://authn.mycomany.com
- Links to the documentation of any relevant APIs, libraries, shared services

Tips: Abstract as much of this as possible and make it easy for your Engineering teams

DATA AT REST

Specify the level of care that must be taken when dealing with sensitive stored data. If encryption is required, specify the algorithm and key bit size. If available, give actionable advice as to how to perform those operations (e.g. supported libraries).

COMPLIANCE AND REGULATIONS

This section is industry and data specific but extremely important. Fines for being found to be out of compliance can be steep and business crippling. As the Software Architect, you are responsible for identifying the areas of the system which fall under the applicable compliance or regulations and how an Engineering team can ensure their application is compliant.

Examples of types of compliance:

- HIPAA
- PCI-DSS
- SOX
- GDPR

OPERATIONS

Before a single line of code is written, there must be a plan for how the team and potentially external operators can observe what's going on in the system. That data can be used to influence the team's backlog (think UX improvement, resource optimizations, feature removal, etc.) as well as the actions to take when faults and errors inevitably occur. This is a necessity to ensure that the identified quality attributes meet or exceed the required specifications (e.g. availability, performance, and so on).

LOGGING AND TELEMETRY

Specify guidelines and/or requirements for application instrumentation here. The types of guidelines you might specify are:

- Logs should have at least three levels of verbosity: INFO, WARN and ERROR
- Logging must be configurable outside of the component's implementation
- Logs should be formatted as serialized objects for ease of querying

MONITORING AND ALERTING

This is unlikely to fall on the Software Architect, but you may find that there are architecturally relevant items you want to call out as they relate to overall developer productivity and/or efficacy.

To get you started, you may want teams to monitor for:

- Failures
- Performance degradation
- Capacity planning
- Intrusion detection

You also might set guidelines for tooling selection. For example, any monitoring tooling must be configurable by a remote API and provide visualizations.

Further, as it relates to business continuity and system availability, you may set thresholds for responding to, and resolving, alerts raised by the monitoring system.

INCIDENT RESPONSE

Like monitoring and alerting, the patterns and practices for incident response will almost certainly fall outside of your immediate scope. However, you'll want to verify and validate that such patterns and practices exist and meet the quality attributes identified. As the Software Architect, you must bring the appropriate teams and departments together when you notice gaps. Raising flags early and helping teams align and prioritize roadmaps is expected of you, even if you aren't directly responsible for the tasks to bridge those gaps.

Relevant books:

- DevOps: A Software Architect's Perspective
- Site Reliability Engineering

APPLICATION DEVELOPMENT MODEL

Precise guidelines and requirements for application development supplies consistency where it matters and autonomy where it's needed. Consistency at the application architecture level provides many benefits. Most notably:

- Source code and documentation is easy to find and read across teams
- Team members can move across teams if desired and ramp up on a new code base quickly, even if it's written in a different language
- Outcomes are predictable enabling the business to make commitments with confidence

PRINCIPLES

Principles are key as they guide our decision making. Here are some application-level principles to get you started:

1. **Integrate early and frequently**
    a. **Description:** All applications should limit work in progress (WIP) and work in small batches, committing to the main branch regularly. No long-lived feature branches
    b. **Rationale:** Early integration reduces risk in overall system delivery by exposing and resolving inherent ambiguities. Frequent integration facilitates the early discovery of implementation issues.
    c. **Implications:** Leverage automated integration tests, which minimize the cost and improve the reliability of integration testing.
2. **Component Versioning**
    a. **Description:** All components must be versioned using the same versioning scheme
    b. **Rationale:** Facilitates communication among components
    c. **Implications:** All component documentation must be versioned and major version changes may not break a contract or change expected behavior

Make it your own but a good rule of thumb is to start with the end in mind and work backwards. The guiding principles which help you move from where you are to where you want to be are perfect candidates for your application development principles.

Tips: Work with Engineers and Engineering Managers/Directors to develop these. You're not looking for consensus, but you do want strong support from some. If you get consensus, that's great but don't be discouraged if you get a lukewarm reaction from some. So long as you have some support and you can show how these principles fit the bigger picture, you can rest assured you have a solid set of application design principles.

DEVELOPMENT PRACTICES

In addition to setting some application development principles, setting some key development practices can help ensure development teams' implementations realize the system architecture (e.g. aligned with the quality attributes). These development practices should be set at a macro level which can be applied to all development teams so stay away from "Use framework [X]" as a practice.

This guide provides some development practices aligned with Lean development principles to help you get started. The following practices are set to enable working in small batches, make work visible, and limit work in progress. See more in The DevOps Handbook (linked in External Resources).

- Work on a short iteration release cycle (2-3 weeks). All iterations should start and end on the same days
- Use git for source control management
- Leverage Test Driven Development (TDD)
- Follow Behavior Driven Development (BDD)
- Design application internals using Domain Driven Design (DDD) techniques
- Use a common Continuous Integration (CI) and Continuous Delivery (CD) pipeline
- Think huge, Design big, Build small

The Software Architect will usually work with Heads of Engineering and/or Directors to best understand the organizational goals from an Enterprise perspective. With that knowledge in hand, you can work backwards to determine which practices align best with those goals.

Tips: While some challenges your organization will face are unique to it, there are some universal industry best practices which should be the foundation of your development practices. Keep in mind that it's a fine line to walk copying high-performing engineering teams and simply performing cargo-cult culture rituals. All your practices should have a purpose with a measurable (positive) impact.

DEPLOYMENT AND HOSTING

Up until now, we've focused on understanding and documenting the problem space and the goals of the business exclusively. Such concerns exist without regard to technology. However, it's finally time to dive into the *solution space* where we structure how the problem is going to be solved.

No matter what you end up building it needs to be served from *somewhere*. Could be a public cloud, private cloud, on-premise or a hybrid setup. The reasons for this vary but it's up to the Software Architect to determine the appropriate hosting location(s) based on the needs of the business. The rest of this document will assume the software will be hosted using a public cloud provider.

VENDORS

Chances are that the vendor has already been selected in your organization, in which case it's usually a fool's errand to migrate away unless there's an extremely compelling reason to. However, if you're starting on a completely greenfield project, vendor selection may fall into your realm of responsibility. In such a case, you may not have all the information needed to properly evaluate vendors—for example, you may be working with data that one cloud provider may have certified compliance in while another does not. With a few exceptions and niche use-cases, any of the three major public cloud providers (AWS, Azure, GCP) will likely be fine but it's up to you to do your due diligence to determine if your business/industry has special needs.

DEPLOYMENT AND RELEASE

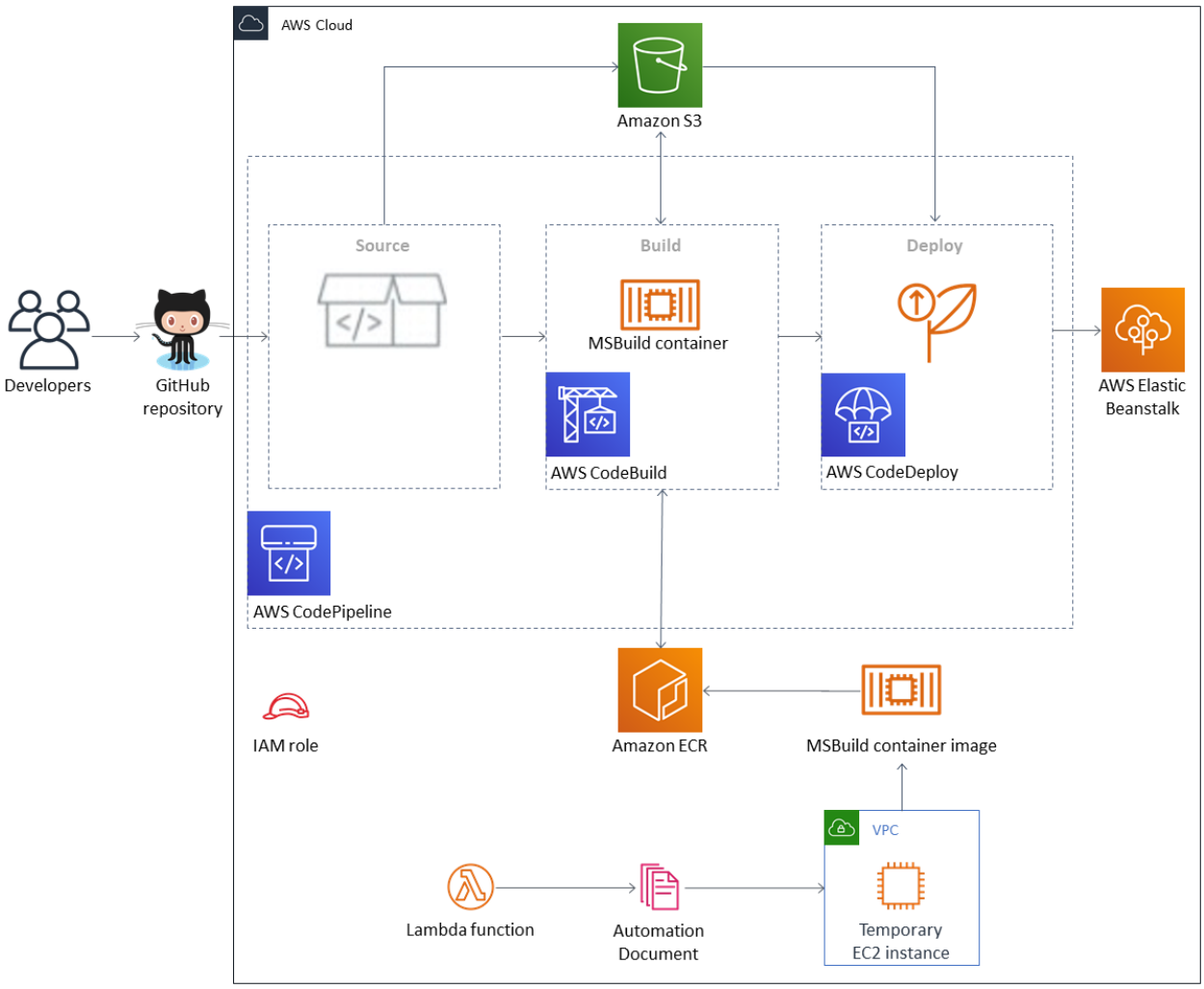
Clearly specify requirements and expectations for deployment and release. Here are some ideas to get you started:

- There must be separate isolated environments for Dev and Production
- Environments must be configured and recreated from code
- All applications must leverage a CI/CD pipeline
- Teams must be able to deploy to production several times per hour
- The system must support self-healing rollbacks
- The system must support multiple deployment strategies including blue/green, rolling and canary

And so on. Like the note about vendor selection, these standards and processes may already be set by your organization. However, this is a key enabler of safe agile development, so these concerns are included in this guide for completeness.

DEPLOYMENT VIEW

Include a runtime view of your system to demonstrate the various technologies, platforms and protocols used to execute your applications. Your deployment view will certainly differ based on your tech stack and environment, but we include one from AWS below as an example:



Source: https://aws.amazon.com/quickstart/architecture/dotnet-cicd-on-aws/

Deployment views can be informal like the one above or use a standard modeling language such as ArchiMate. I prefer the former with the caveat that consistency should be enforced (e.g. icon sets, keys, colors, etc.). Your diagrams exist to tell a story and achieve shared understanding so pick a style and keep it.

Tips: Your organization is probably not Netflix but that doesn't mean you should be doing risky and error prone XCOPY deployments, either. Stand on the shoulders of giants as many of these common problems have already been solved and been abstracted away for you to consume directly. Seek to provide self-service platforms to the development teams whenever possible.

Relevant books: The DevOps Handbook, Accelerate

TECHNOLOGY DECISIONS

When faced with an architecture decision, using a template-driven approach provides a consistent approach to communicating the decision and thought process which led to that decision. Further, it adds traceability to decisions and with your favorite project management tool, can be linked to work items, user stories, etc.

PURPOSE

Used for communicating decisions, their justification, and providing auditability of decision making.

TEMPLATE

Feel free to create your own template, but here's an example of a template which can be used directly:

ID	**[Unique Identifier for this Decision]**
Subject Area	Examples: Security, API, User Interface, etc.
Title	[Descriptive name for this decision. Example: Message Queue or Authentication Protocol]
Problem Statement	[Expanding on the Title above, provide a descriptive and thorough problem statement]
Guiding Principles	[What principles, assumptions and constraints will use to decide from the considered options? For example: Reduce operating expenses, Minimize impact of change to legacy systems, etc.]
Drivers	[What's driving this decision? Example: an Enterprise initiative, a particular team, etc.]
Options	[All the available options being considered in this decision]
Decision	[The decision from the available options]
Justification	[The rationale and thought process which ultimately led to the decision]
Implications	[Any implications this decision has outside of its immediate scope. For example: You may call out integrated systems which will need to change or an increase in traffic to a particular area of the system]

DECISION LOG

All of the relevant decisions made should either be contained or (ideally) dynamically linked within your software architecture documentation.

EXTRAS

Approaching your system design using the above techniques will provide your teams and stakeholders with all the information, guidance and tools they need to meet the business commitments on time, quality and budget. This is miles ahead of an "architecture by accident" design but you may be asking yourself *how* can you arrive at the deliverables above? Recall that the architecture of a system is not about the diagrams or other artifacts, but rather the critical decisions made which enable teams to realize the quality attributes and deliver customer value.

This section provides key resources which will help you make those critical decisions and help you on your journey to becoming a skilled software architect.

HOW LONG DOES THIS TAKE?

The typical Architect answer is "it depends" and that's the case here too. However, the time it takes to provide a system design is a bit nuanced. To create the artifacts and make the decisions is a relatively short process (even large systems can be documented within an uninterrupted week). The true source of work, and hence time, is in understanding the problem space by conducting interviews, leading/guiding strategic design sessions, etc.

Luckily, such collaborative design sessions should be a part of the engineering culture, and if it's not, it's your responsibility to campaign to make it so. So, you get to trick everyone in a sense. You have a vested interest in understanding and documenting the problem space and stakeholder concerns. Since everyone is benefiting from the activities by achieving shared understanding and strengthening rapport, they happily participate. Everyone is helping you with the *first step* of software architecture—understanding the business need and customer value and they don't even know it! (Unless they read this guide 😐)

The amount of time you spend "doing architecture" is just synthesizing all the data collected from your stakeholders to present in various views depending on concern. Which, with practice, becomes an innate ability. Armed with experience, reference architectures and standard patterns, you'll be able to create views for any level of audience.

PATTERNS

If one does not exist, establish a pattern and reference architecture catalog for your organization. Nothing fancy is required here but ideally this content would live in some sort of Software Architecture wiki. Do not underestimate the power of a pattern catalog. It should provide the following benefits:

- All system designs created by software architects reference existing (ideally tested) reusable patterns using common terminology and technology
- Allows you to distribute your impact across teams by discouraging the introduction of an "architect gate". Remember: as software architects our mission is to provide value and get out of the way

What should you include in your patterns and reference architectures? Each item in your catalog should have at a minimum the following:

- Clear title, description and category
- Clear applicable scenario(s)
- Diagram – interaction, component, etc.
- Pros, Cons, and Tradeoffs made when selecting this pattern over similar patterns

- Links to alternative patterns for the applicable scenarios

You may also want to put restrictions or recommendations on the technologies used to implement those patterns (e.g. use SNS/SQS for service to service messaging). Check out the *Cloud Design Patterns* link in the Web Resources below for some ideas of what to include in your catalog.

Tips: Just like application design patterns (think Gang of Four), you don't win by applying every pattern under the sun. Your catalog must guide teams to selecting the *appropriate* design pattern based on the scenarios and constraints present.

EXTERNAL RESOURCES

Must read books (no particular order):

- The DevOps Handbook
- Accelerate
- Release It!
- Implementing Service Level Objectives
- Designing Software Architectures
- Domain-Driven Design Distilled
- The Phoenix Project
- The Unicorn Project
- Antifragile

Web Resources

- Cloud Design Patterns
- AWS Well-Architected Framework
- Netflix Tech Blog
- Chris Richardson's Blog
- Software Engineering Institute at Carnegie Mellon University
- TOGAF

LICENSING

I'll be the first to admit there's nothing groundbreaking in this guide but there is something to be said about the value in packaging everything up for easy consumption. Calculus hasn't changed in the last 400 years, but some textbooks are more successful at making it all "click", right?

You're free to use, link, distribute, copy/paste, and reword any content found in this guide however you wish. But if you found this content is helping you become a more skilled, competent and confident software architect, the following is appreciated:

- **Attribution and link to my LinkedIn**
- **Donate to your local elementary school or animal shelter. Teachers always need supplies and animals always need adopting.**

THANKS AND GOOD LUCK!