**Chapter 3. What’s the Pattern? Architecture, Pattern Languages, and Design**

*This chapter is split into two sections. The first introduces Christopher Alexander’s groundbreaking work on evolutionary growth versus big-bang projects. We also look at software design patterns, discuss the need for a pattern language specific to cloud native, and explain why having a pattern language increases the chance of a quick and successful transformation. In the second part, we discuss patterns themselves, how they are made, how they work—and our process for creating cloud native patterns for this book.*

Architecture: “The complex or carefully designed structure of something.”

There is a reason that we apply the term “architecture” to both buildings and computer systems. The term describes a non-trivial system whose shape has come about through careful thought, imagination, and planning—also known as design.

In his book *A Pattern Language*, first published in 1977, Christopher Alexander consolidated decades of intelligent observation of architecture. He thoughtfully analyzed what works—and what does not—when we undertake to design and build human systems. Alexander points out that “right” architecture has nothing to do with classical beauty. Rather, it creates a structure that is comfortable and easy to use and maintain. It’s one that feels like a responsive, even living, place that supports and extends those who use it rather than constraining or frustrating them.

Alexander’s quality of “rightness” is difficult to quantify or describe yet innately recognized when we experience it. We have all seen houses or cities that feel right, that we’d like to live in, just as we’ve been in ones that simply feel wrong. A gorgeous city square lacking shade from the sun and offering awkward seating, or none at all, does not encourage visitors to linger. A courtyard with a mixture of sunshine and shade, offering trees and fountains and comfortable places to sit, is a far more human venue—a “right” place people will be happy to spend their time in and work to maintain.

As software engineers we have all experienced, in the same way, computer systems and teams that are not “right”—brittle systems that constrain and frustrate us, that fail to support us as users or maintainers. Hopefully we’ve also worked on systems that made us proud and that were a genuine pleasure to develop and grow, and collaborated with teams that feel like a circle of friends who happen to share high productivity and a passion for clean code.

But how do we make the good stuff happen? How do we move from instinct to actuality?

**Patterns**

Alexander’s key to building “right” architecture was the creation of context-specific patterns that were easy to explain and teach and were above all deliverable. He developed these patterns over a long career spent observing what did and did not work on the ground.

The pattern languages Alexander developed cover all levels of design, from micro to macro. At the micro end of the spectrum he describes individual spaces and how to make them comfortably habitable and instinctively “right” using templates such as “Pattern 128: Indoor Sunlight” and “Pattern 236: Windows Which Open Wide.” Alexander slowly widens his lens from single rooms to whole buildings and the relationship between them, then outward further to towns, cities, and entire geographic regions. These macro patterns include planning for both urban and rural settings since different contexts produce different needs. Examples like “Pattern 11: Local Transport Areas” and “Pattern 52: Network of Paths and Cars” show how broadly scoped patterns can be applied to benefit large populations, not just single dwellings.

*The fact is that very few things have so much effect on the feeling inside a room as the sun shining into it.*

*Pattern 128: Indoor Sunlight*

Collectively the patterns serve as universal guiding principles for design. “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” (Christopher Alexander, *A Pattern Language*, 1977).

So when does an actual pattern language emerge from all these building blocks? Individual patterns themselves are not a language, but more like vocabulary terms. A group of individual patterns can be put together to describe a vocabulary (still not a language). A pattern language, though, presents the patterns along with guidance on how to construct solutions (e.g., A follows B, or C may follow from B in certain cases). Once you have these “grammar” rules in addition to the vocabulary, then you have a language.

The notion of patterns for constructing buildings and even towns was instantly embraced by architects, and more than 40 years later *A Pattern Language* remains in print. The concept also piqued the interest of computer scientists, who recognized the value of applying similarly concise, specific, and reusable solutions within the field of software development.

In 1987 two software engineers, Ward Cunningham and Kent Beck, inspired by Alexander’s work, presented a paper on applying the idea of design patterns to computer programs. This was the catalyst for a whole new pattern-based philosophy of software design, and the new school of thought attracted many converts. The software design patterns movement crossed over to the mainstream in 1994 when the Gang of Four—software engineers and researchers Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—published  *Design Patterns: Elements of Reusable Object-Oriented Software.* A quarter of a century later, this hugely influential book can still be found in the personal reference libraries of many a software developer.

The appeal is obvious: a software design pattern, just like an architectural pattern, is a reusable solution to a commonly recurring predicament. A software design pattern is *not* a finished design directly translatable into code, but a platform-agnostic description of how to solve a specific problem. It also functions as a language, one that helps developers to communicate better since they can now name the important things and agree on basic principles. Having this common language grants a shared understanding that allows devs to save time while elevating discussions to a higher level, because the basics have been agreed upon and settled.

Cloud native itself is a particular approach to software design, a paradigm shift to a new way of designing, building, and running computer applications. It is based on cloud Infrastructure-as-a-Service combined with new operational tools and techniques like continuous integration, container engines, and orchestrators.

The objective is, almost always, to improve speed. Companies can no longer simply deliver a product or service. To survive and prosper in the new age of instant information in the palms of our hands, they must also help customers seamlessly use, consume, or purchase their wares. You can no longer be “just” a healthcare or financial services or retail enterprise: like it or not, realize it or not, all companies are now software companies.

Existing software design patterns may still be useful to cloud native systems so long as the different context is carefully considered, though architectural patterns certainly do not apply. Thanks to the aforementioned paradigm shift, new patterns have been emerging to address the new cloud native architecture.

When Container Solutions was founded in 2014, we didn’t even have a name for this new approach to developing software applications, much less a shared understanding of common scenarios and effective methods. Cloud native, as we now call it, has reached the point where we can create a new pattern language allowing us to name—and, of course, implement—optimal context-specific practices in this rapidly evolving technology.

**A Time for Design**

Having established the benefits of cloud native architecture, you might now reasonably expect the assertion that cloud native systems are intrinsically “right.” We would love to say this, but unfortunately it isn’t true. Cloud native systems, no matter how robustly containerized, microserviced, and orchestrated they might be, are no more likely to exhibit rightness than any other computing architecture. In fact, in some cases, they can be worse.

This is because there are times where cloud native complexity is simply not called for. Highly stable and proficient systems that require little change or updating, for example, make for ideal monoliths and should be run as such. Attempting to split such a system into containerized microservices would be an interesting thought experiment, costing time and resources to do while ultimately producing little functional improvement. “Monolith” is not a dirty word. It is simply another kind of architecture, and there are some contexts where it is absolutely the right approach.

Most of the time, though, given the complexity of modern software and the sky-high expectations of users, cloud native is likely the right approach. Unfortunately this is a highly complex process; while almost all transformation initiatives start off well, many quickly go wrong. Why does this happen?

Cloud native can be deceptively easy to implement at first. Anyone with a credit card can log onto a public cloud provider and have an initial instance up and running in a matter of hours. And it is indeed true that the first few experiments with cloud native technology, typically in a careful testing-the-waters context, are almost always easy and successful. This unfortunately creates the hopeful illusion that full migration to cloud native will be equally easy: just install a few tools and go live! However, implementing full-scale enterprise cloud native is in fact very difficult due to the complexity of distributed systems, which increases exponentially with scale.

This next step, after the easy first experiment, is in fact where cloud native migrations usually go wrong. This is a very new technology, and there simply is not enough knowledge within most organizations to successfully navigate cloud native’s complexities. And currently, given the immature state of the ecosystem, it is not yet possible to go forth and purchase one full-service solution that can simply “make you” cloud native.

This is also, however, the right time to begin constructing a set of sound strategies and design practices in order to facilitate cloud native migrations. We are far enough along in the evolution to understand at least the basics of what works, what does not, and why.

Even once you have successfully transitioned to cloud native and are enjoying all the benefits of this latest and greatest technology, you don’t get to simply stop. There will eventually come a day, probably sooner rather than later given the accelerating pace of technological advancement, when a new technology or paradigm emerges to change the rules yet again. So, ultimately, what we need is not best practices for adopting cloud native or Kubernetes. The most enduring and valuable thing we can take from a transformation is the ability to change and adapt to new circumstances, whatever they are. The foundation of cloud native philosophy is evolving right along with your environment through constant small innovations and experiments. This is how we de-risk change: by building continual evolution and learning into the architecture itself. If you do it all the time, suddenly change is no longer scary.

What we now require is a way to identify, describe, and name these evolution-friendly strategies and practices—all the better to share our hard-earned knowledge and help organizations create “right” cloud native systems. A cloud native pattern language does exactly that.

**A Cloud Native Pattern Language**

Alexander wrote that “each pattern can exist in the world only to the extent that it is supported by other patterns.” A design is incomplete if it doesn’t contain mutually supportive patterns to address the entirety of the system, from the largest problem to the most detailed. In addition, he concluded the overall vision cannot be delivered in one big bang, but must grow toward the envisioned ideal.

Sound familiar? Alexander’s pattern parameters correlate quite well with the cloud native principles of interrelated microservices, collaboratively developed and delivered in small, iterative steps toward the overall goal.

Just as Alexander’s building patterns must interrelate and support each other, so must the patterns of cloud native architecture. For example, a “Distributed System” may depend upon “Microservices Architecture,” “Containerized Applications,” and “Dynamic Scheduling” while existing within a larger context/goal, such as a “Highly Secure System.”

The point of a pattern language is to make it easier for developers to discuss, learn, and apply the best practices in cloud native. In this book we will share some of the patterns that we’ve learned from over five years of helping enterprises move to cloud native.

The patterns themselves come from experience working with our clients and with cloud native itself, but they also pull in years of reading books and articles in every conceivable related area. Other patterns publications, management and strategy books, and seminal papers like *Manifesto for Agile Software Development* and Mintzberg’s *Of Strategies, Deliberate and Emergent* all fed our understanding and helped form the thoughts in the book.

Patterns inherently build upon other patterns; that is how patterns and pattern languages work. Similarly, we the authors are indebted to the work of many who came before us. Since it is impossible for us to acknowledge every one of the countless influences and sources of inspiration that fed into our own work, however, we elected to cite sources only when directly referenced or quoted.

**Patterns in Context**

Cloud native systems are innately complex. This means having a high likelihood of failing to deliver even the table stakes of good system design: being stable, efficient, and fit for their purpose. Sometimes one cloud native approach might be better than another—or perhaps a different architecture entirely could be the superior choice. There is no one design that will work well in every circumstance, and so design patterns must be context-specific. A design that ignores context will almost certainly be a painful one to deliver and difficult to live with—if it works at all.

What kind of contexts should we consider when making software design choices? There are a lot! For example:

* The existing skills of your teams
* The time frame and goals of your project
* The internal political situation (how much buy-in exists for the project)
* Budgets
* Legacy products and tools
* Existing infrastructure
* Emotional or commercial tie-in to vendors or products
* Ongoing maintenance preference

The right choice almost always depends on where you are now and where you want to be. This is as true in designing, say, distributed system architecture as it is in designing a townhouse.

As an example, one effective and popular cloud native pattern is Dynamic Scheduling—that is, the use of a container orchestrator like Kubernetes to handle the deployment and live management of applications on your physical infrastructure. Though dynamic scheduling is great in most contexts, there are some where it can be disastrous. Nothing is perfect in every environment, Kubernetes included. Let’s say you decide, as part of your transformation, to apply the Lift & Shift at the End pattern to move one or more of your existing VMs straight onto the cloud, without refactoring. VMs are designed to run in highly stable environments, though, and so require either re-architecting or some kind of wrapper to shield them from the highly animated environment of a cloud native platform where they could be switched on and off randomly as load demand requires. Your team doesn’t really know how to handle orchestration yet, though, so if you encase your VMs inside Kubernetes without teaching the engineers how to maintain them, what you have is a pure (and totally unnecessary) disaster in the making. And it happens not due to the Dynamic Scheduling pattern itself, but the context where it gets applied.

This is why it is vitally important that, before selecting which cloud native patterns to implement, enterprises first understand the current context for their organizational needs and aspirations as well as identifying the proper target context. But how to pragmatically assess these contexts?

A tool was required and, in the course of guiding companies of all sizes over a wide spectrum of sectors (from financial services to manufacturing to healthcare, to name but a few), our cloud native engineers developed one. The Container Solutions cloud native Maturity Matrix is an assessment tool for helping enterprises understand where they are now as well as identifying where they want to go: their existing versus desired contexts. The Maturity Matrix is discussed in depth in [Chapter 6](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch06.html#tools_for_understanding_and_using).

It is important to note that not all cloud native contexts are concerned with technology. Migrations are not just about software; psychological and social forces within the organization heavily influence success or failure as well. The context of an organization’s management process, team structure, and internal culture must be assessed before assigning patterns. These human-centered context areas are as important as tech-centered ones like provisioning and maintenance/automation (see [Chapter 2](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch02.html#the_human_challenge_of_cloud_native)).

**Cloud Native Context in the Real World**

Continuous integration (CI) is an example of a high-level pattern that seems to nearly always apply in the cloud native context. Within any developer’s codebase, the more it diverges from the main code branch, the harder it becomes to merge working changes back into the mainline—eventually, the merge may even become impossible. This seriously reduces a system’s stability and availability. CI calls for putting in place tooling that makes it very easy for devs to merge their changes back into the main codebase, on a daily or even more frequent basis, while simultaneously checking that no bugs have been introduced.

There are contexts, though, where CI would be less useful. When there is a single developer and a single codebase, for example, an automatic merging tool provides less value. A context where CI would be not just useless but actively bad as a pattern is in a situation where there is no automated testing. If everyone is merging insufficiently tested changes into the main codebase, you soon end up with a malfunctioning mess.

Continuous integration and continuous delivery (CD) use automation to ensure that new application code is always tested, secure, and ready for deployment. The automation and stability provided by CI/CD are so essential to cloud native architecture that few, if any, enterprise cloud migrations would choose to *not* adopt this pattern—as long as their existing technological context includes thorough automated tests.

There is, however, another nontechnological prerequisite context for CI/CD to be adopted successfully, and that is executive commitment: the culture context. These projects are difficult and expensive. They require Executive Commitment, one of the core organizational patterns we will discuss later. If you tried to implement CI/CD in an enterprise without the cultural context of senior-level support, you would quickly run out of corporate patience and budget.

So CI and CD are both key cloud native patterns that vitally depend on context. No matter how fundamental to the architecture, if they are executed in the wrong circumstances, they will fail. This shows us why “the right pattern” can only be the right pattern *in the right context*.

**Common Contexts**

Many companies looking to commence a migration to cloud native share a consistent and typical setup. Often, they are organized in traditional Waterfall style, with perhaps some transition to Agile (typically, in their use of Scrum), and have:

* Traditional Waterfall process with deliveries every few months
* Monolithic applications
* Older languages (typically Java/C#, but sometimes even Cobol)
* Strong, inflexible hierarchy
* Little or no automation of infrastructure and development processes

WealthGrid, our example enterprise, is an extremely typical example. For a long time WealthGrid found success in its traditional financial services market by delivering reasonable updates, fairly often. However, it can take up to six months for a new feature to reach customers. This makes it pretty much impossible to incorporate user feedback and requests in a useful time frame, meaning that WealthGrid instead has to more or less guess in advance what customers want—and hope that they guess right.

Now organizations like WealthGrid have realized they need to compete with new, web-based companies such as contender banks—mobile-only banks providing full-service accounts solely accessed from Android and iOS devices. It’s not only financial companies finding themselves under pressure to keep up with tech-savvy competitors. Perhaps they are online retailers led by customer expectations of near-instant feature updates and a highly personalized user experience. No matter the business model, fast innovation has become essential for survival in a marketplace that is becoming increasingly global yet customized. These days, thanks to technology, any company can compete in any market—Amazon can become your hometown brick and mortar grocery store—and do so while providing each customer with personally tailored services.

Simply put, a high level of customized services, delivered with little or no downtime, is what customers now expect. This means that, for a business, velocity, time to market, and ease of innovation are more important than cost saving.

Staying abreast with the newest technology in order to provide the same cutting-edge user experience as the competition requires attracting developers who want (and are able) to work with the very newest tech. Since modern applications tend to be distributed, this is a competitive hiring arena.

Given that these enterprises, no matter what their core business, all originate from similar circumstances and are driven by similar needs, it is not surprising that they face similar difficulties:

* Decisions are typically made according to existing practices. This is appropriate for stable tech ecosystems like virtual machines. In cloud native, though, currently there are few established practices.
* In a traditional hierarchy, the top managers decide, but they don’t fully understand the complexity of building and maintaining distributed systems—and therefore allocate insufficient resources.
* For the next three to five years, cloud native will still require a lot of investment due to the technology’s relative immaturity.
* Dealing with immature tech requires more experimentation and research than straightforward project management typically allows. Processes such as Scrum create problems in particular by placing massive pressure on delivery, which reduces resources for research and innovation.
* Large enterprises are optimized to preserve the status quo and embrace change slowly and reluctantly, while cloud native requires quick changes and the ability to work in an ambiguous environment. Such organizations must undergo a fundamental culture shift.
* There is not enough cloud native knowledge in enterprises, or indeed in the current tech sector overall, to support effective widespread migration. Companies don’t even know how much they do not know.

Successful cloud native transformation requires most enterprises to make changes in both technology and internal processes and culture in order to prepare the path. As a straight-up technology, due to the complexity of distributed systems, full-scale cloud native is very difficult to implement. When coupled with organizational and cultural challenges, the way forward can seem formidable indeed.

When the right patterns are applied in the proper context, the results can be powerful. Time is saved. Money is saved. People get to work in cloud implementations that feel “right”—for both the enterprise’s business mission and the humans who work to execute it.

**Patterns, Pattern Languages, and Designs**

*Here we get deeper into pattern languages and explain the details of good patterns: how they work, how to create them, and what is the best way to use them.*

Originally, Alexander developed pattern languages to bridge the gap between professional architects and citizens in the community so that ordinary people could design their own dwellings. Software design patterns subsequently arose for constructing non-concrete objects: computer programs. In both cases these patterns allow us to fill technical gaps between expert practitioners and those with less experience. When junior developers can read, understand, and apply patterns created by experienced engineers, this greatly accelerates the production of new software/systems. Software-design patterns can speed up the development process by providing tested, proven development paradigms.

Even more importantly, though, effective software design requires awareness of conflicts and issues that may not become apparent until much later in the implementation. With patterns, the juniors benefit from the design knowledge that the more advanced practitioners worked hard to gain—without needing to spend years building the deep understanding of the subtleties of contextual relativity that makes this kind of foresight possible.

When it comes to cloud native, most of us are junior. The technology is so new, and our understanding of the architecture so nascent, that sharing our rapidly evolving knowledge is especially valuable. Creating cloud native–specific patterns is a way to name the things we are learning so developers and engineers can communicate using common names for software interactions. As we learn and continue, the design patterns will evolve and improve alongside our understanding.

**How Patterns Work**

Patterns are not a hack. That is, they are not a quick and easy way to solve difficult problems without careful thought. (In fact, that’s where the application of patterns often fails: because people try to treat patterns, particularly the Gang of Four patterns, in just this way.)

Instead, patterns are a language for sharing context-specific working solutions. Context comes in when we select and fit together patterns to form a design—which brings us to the three core concepts at hand: patterns, pattern languages, and design.

* **Pattern:** A contained and specific approach to addressing one particular, commonly occurring need. A pattern is designed to be re-usable and applicable whenever the appropriate context occurs. There are many ways to implement each pattern, depending on different circumstances.
* **Pattern language:** The method of describing patterns, their useful organization, and good design practices within a particular field of expertise. Alexander had architecture, which begat software design patterns, which in turn spawned pattern languages in hundreds of abstract areas, including disaster prevention, innovation, cooking, and even a pattern language for writing patterns.
* **Design:** Patterns fitted together to define a full system, or perhaps one part of a larger system, with some guidance for selecting and applying them in a particular order. Designs provide general solutions, a sort of road map even, documented in a format that doesn’t require specifics tied to a particular problem.

These terms vary based on which patterns are being applied and how. In the context of Alexander’s building patterns we use “patterns,” “pattern languages,” and “designs” to construct, from micro to macro scope, individual rooms, buildings, neighborhoods, cities, regions, and so on. Gang of Four and other software-related patterns, on the other hand, pretty much only discuss “design patterns” with no concept of “language” or “design” as a system for putting them together. For the purpose of this book, we are taking the Alexandrian approach; we have “patterns” to form a “cloud native pattern language” and “designs” of typical cloud native transformations.

For example, the cloud native pattern “Distributed System” may depend upon “Microservices Architecture,” “Containerized Applications,” and “Dynamic Scheduling”. Together they can also form part of an overarching design, such as a “Highly Secure System.” We’ll describe these specific cloud native patterns and present sample designs in future chapters. For now, let’s talk about what goes into creating a good pattern of any kind.

**What Makes a Pattern?**

Taken together, patterns, pattern languages, and design form the way to simply but fully describe a complex system. However, there is no one right—or, for that matter, wrong—way of creating and presenting patterns.

Fundamentally, however, all patterns convey the same areas of information:

* A *title* that immediately and concisely conveys the problem being addressed
* A *context* statement describing the problem at hand
* *Forces* that influence the problem and make it difficult to solve
* A *solution* (or solutions) presented in detail
* The *consequences* of applying the solution

A good pattern is precise and concise. It concerns itself with a very specific and limited problem, isolates and identifies the forces that shape that problem, and tells us what to do about it. It ends by describing the new context that will be created once the problem is addressed. And it does all this in simple language accessible to almost any reader—you do not need deep expert knowledge to understand a pattern. The whole idea is to make deep understanding beside the point; we are trusting in the experience and knowledge of the pattern writers to create a guided and reliable path for us to follow.

How do patterns get made? Quite honestly, just as there is no “official” template for presenting a pattern, there is no authority in charge of creating—or overseeing the creation of—pattern languages.

The closest thing the patterns world has to a governing body is the Hillside Group, an educational nonprofit organization based in the United States. It was founded in 1993 to organize the efforts of various thought leaders in the then-emerging software design patterns movement to converge their separate efforts toward defining and documenting common development and design problems into a pattern language.

The Hillside Group functions as the de facto hub of the global patterns community, which gathers at annual Pattern Languages of Programs (PLoP) conferences that Hillside sponsors. The conferences focus on pattern writing workshops and invited talks related to pattern development. One frequent PLoP conference presenter is Takashi Iba, a prominent patterns researcher at Keio University in Japan. Iba is also a prolific patterns author who, working with his students over the past decade, has created more than 20 pattern languages containing more than 600 patterns covering many different topics. As a part of this process Iba has developed his own process for creating patterns and a protocol for presenting them.

**How (Iba) Patterns Are Made**

When developing a new pattern language, Iba discovered that it is important to “mine the seeds” of the patterns from real-world examples of optimal experiences or best practices. He named this approach “pattern mining.”

*In the pattern mining, first, miners explore their experiences, observations, episodes, or documented past work related to the subject at hand. Through this exploration, they look for and identify hidden knowledge used for the target. This knowledge may include associated rules, methods, tips, or customs. Next, the miner finds critical connections among these related items so that prospective pattern begins to form a meaningful whole.*

*Takashi Iba, “Pattern Language 3.0 and Fundamental Behavioral Properties,” keynote address, PURPLSOC Conference 2015*

In Iba’s pattern mining process, group members start by identifying a subject and then brainstorming any and every rule, custom, insight, or method they deem to be important about the subject. These are all written down on sticky notes. They briefly describe each insight to the group and then add its sticky note onto a large sheet of paper. Everyone in the group brainstorms ideas and takes turns talking about them until no one has any more thoughts to share. At that point, they examine the mosaic of sticky notes and collaborate to organize similar ideas into groups. The groupings that emerge from this process are the “seeds” of potential patterns.

The group approach is important for creating the most accurate and incisive definition of a problem. As humans, we are all subject to cognitive biases and wrong assumptions (see [Chapter 4](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch04.html#beyond_patterns_behavior_biases) for more about behavior and biases); eliciting multiple points of view helps authors to avoid patterns becoming skewed by individual prejudices or cognitive blind spots.

**Presentation**

The next step is to write up the pattern seeds into a formal pattern format. An Iba-style pattern states the essential elements of title, context, forces, and solution very simply, and preferably in no more than one page of text. In addition, an Iba-style pattern will also contain a simple diagram.

The format is presented in three sections. The first consists of the pattern name, introduction, and illustration; these three elements serve to provide a summary of the pattern.

* The *Pattern Name* is a short, memorable name that concisely and accurately describes the pattern.
* It is followed by the *Introduction***,** a one-sentence executive summary of the problem that the pattern addresses.
* The *Illustration* comes next to provides a visual representation of the pattern’s core concept.

The second section provides more details about the problem being addressed and presents the pattern’s context, problem, and forces:

* *Context* describes when and under which circumstances the pattern should be used.
* *Problem* is the undesired but common consequence that typically arises in this context.
* *Forces* are unavoidable factors that influence the problem and make it difficult to overcome.

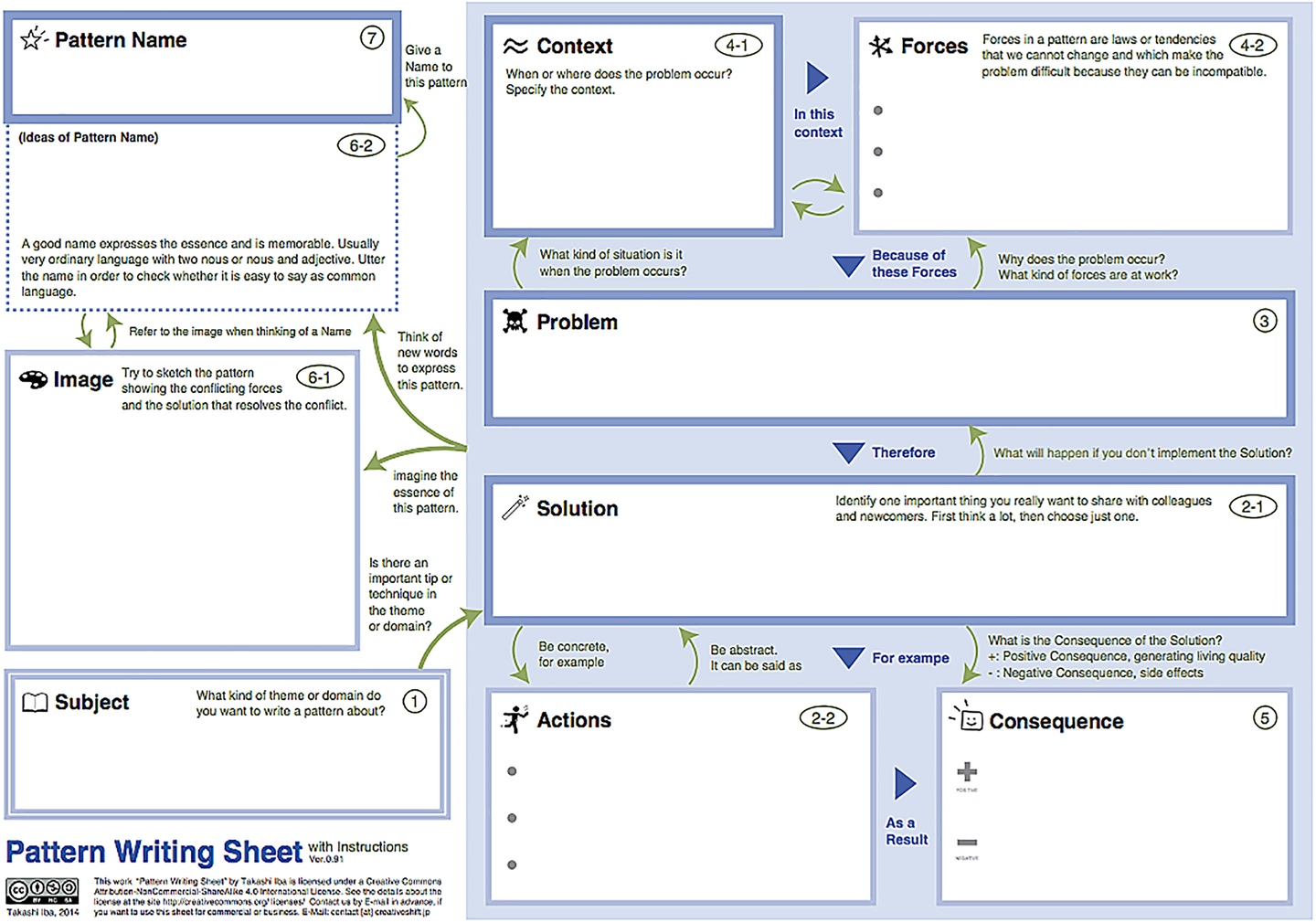
The final section presents the solution, related actions, and consequences of applying them.

* The *Solution* is one way to resolve the problem.
* *Actions* provide some concrete, practical steps to address the situation.
* *Consequences* describe the outcome when this pattern is applied to the problem.

As an aid to envisioning and articulating patterns, Iba also created a Pattern Writing Sheet, which he has made freely available under Creative Commons licensing. [Figure 3-1](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch03.html#takashi_iba_apostrophe_s_pattern_writing) shows the template, which contains instructions for identifying elements of a pattern and spaces for defining the Context, Forces, Solution, and so on.

**Our Approach**

The authors of this book followed largely in Iba’s footsteps. Some patterns were mined during whiteboard sessions with Container Solutions engineers, others during collaborative brainstorming and epic patterns-writing sessions in London, Amsterdam, and Baltimore. Early versions of six patterns were produced as part of a paper presented at the 2018 PLoP conference. That paper, titled “A Cloud Native Transformation Pattern Language,” inspired the creation of many more patterns and eventually grew into this book.



**Figure 3-1. Takashi Iba’s Pattern Writing Sheet**

We have, however, expanded upon Iba’s minimalist format. The fundamental components of Context, Problem, and Solution are all there, but explained in greater depth than true Iba style would dictate. We have also listed related patterns and added common pitfalls related to the pattern that we have observed occurring during actual cloud migrations. We are also going to include any cognitive biases that contribute to the problems described in the patterns or that are a basis for solution nudges. All of this is an effort to help the reader contemplating their own cloud native transformation project.

As is typical, the patterns are presented as a library for readers to draw from. However, we also wanted to present them in the vibrant, living context of a typical migration initiative. So we came up with the story of WealthGrid as a way to bring them to life. We hope you will find yourself in this story, imagine yourself a bank, a travel agency, whatever your business, suddenly finding yourself under pressure from a radical new competitor. You need to evolve by building a new way to deliver your services or products, or else this new paradigm or company entering your sector is going to drive you out of business. The effective way to adapt quickly is to use a pattern design as the roadmap to your new system.

Patterns are the foundation layer of this solution, but they are not the story itself. Think of them more like Legos, interchangeable building blocks that can fit together in any number of ways. We are giving you the pieces you need to put together for designing, creating, and envisioning your own custom solution.

You don’t need to know much, if anything, about the cloud to use cloud native patterns. Patterns give you the ability to describe your own ideas and solutions for your new home to your architect, but that doesn’t mean you have to build the house yourself. Patterns are just a means to express yourself. They don’t tell you exactly how it’s going to be done—they describe the scaffolding you will work inside to build the solution that exactly fits you, in whatever context you inhabit.

New cloud native patterns are emerging constantly. To continue sharing them and extending the cloud native pattern language we have established here, please visit [www.CNpatterns.org](http://www.cnpatterns.org/).

This is where to find the latest pattern developments, but also an online community for discussing and creating new patterns. We are inviting people from across the industry, thought leaders and influencers but most importantly everyday engineers and managers—those out there working elbows-deep in cloud native code and architecture—to contribute and participate. Hope to see you there!