Chapter 23. Diagramming Architecture

Newly minted software architects are often surprised at how varied the job is, outside the technical knowledge and experience that brought them into the role to begin with. In particular, effective communication is critical to an architect's success. No matter how brilliant your technical ideas, if you can't convince managers to fund those ideas and developers to build them, your brilliance will never manifest.

Diagramming is a critical communication skills for architects. While entire books exist about each topic, we'll hit some particular highlights for each.

To visually describe an architecture, the architect often must show different views: for example, they might start with an overview of the entire topology, then drill down into the design details of specific parts of the architecture. However, showing a portion without indicating its place within the overall architecture will confuse viewers.

Representational consistency is the practice of always showing the relationships between parts of an architecture before changing views, and it's equally important in diagrams and in presentations. For example, if you wanted to describe the details of how the plug-ins relate to one another in the Silicon Sandwiches solution, you'd start with an architecture diagram showing the system's entire topology, then the relationship between it and the plug-in structure, before going into the plug-in structure itself. Figure 23-1 provides an example.

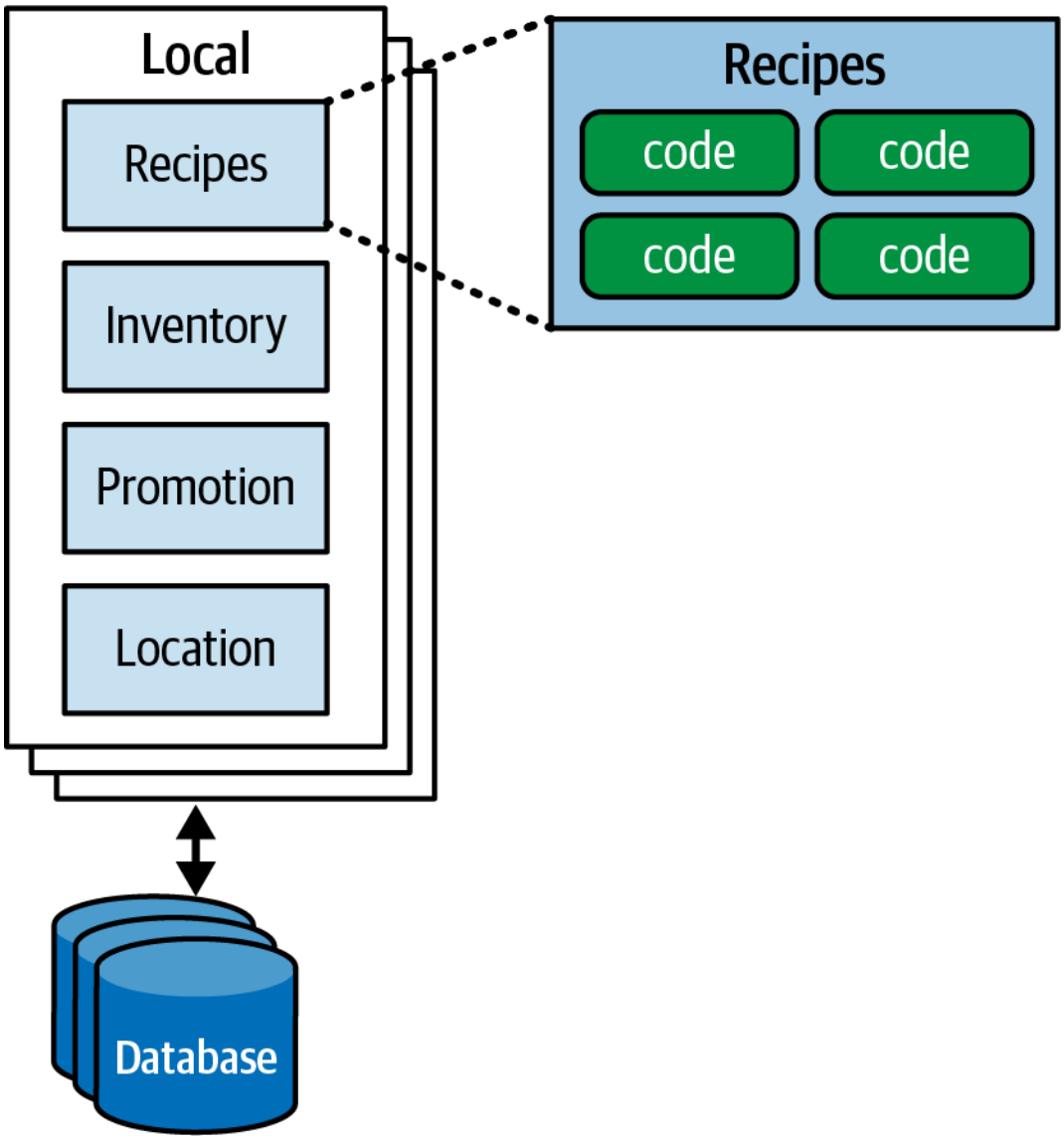

Figure 23-1. Using representational consistency to indicate context in a larger diagram

Use representational consistency carefully to ensure that viewers understand the scope of the items being presented and eliminate a common source of confusion.

Diagramming

The topology of an architecture is always of interest to architects and developers because it captures how the structure fits together, and allows teams to form a valuable shared understanding. We encourage all architects to hone their diagramming skills to razor sharpness.

Tools

The current generation of diagramming tools for architects is extremely powerful, and every architect should learn their tool of choice deeply. But don't neglect low-fidelity artifacts, especially early in the design process. Building very ephemeral design artifacts early prevents architects from becoming overly attached to what they have created, an antipattern we call *Irrational Artifact Attachment*.

Irrational Artifact Attachment

The Irrational Artifact Attachment antipattern describes the proportional relationship between a person's irrational attachment to some artifact and how long it took the person to produce that artifact. If an architect spends four hours creating a beautiful diagram in some tool like Visio, they'll be even more irrationally attached to that artifact than if they'd invested only two hours.

One benefit of the Agile approach to software development is that it involves creating "just-in-time" artifacts with as little ceremony and ritual as possible. This is one reason so many Agilists love index cards and sticky notes: using low-tech tools lets people throw away what's not right, freeing them to experiment and allow the true nature of the artifact to emerge through revision, collaboration, and discussion.

The classic ephemeral artifact is a cell-phone photo of a whiteboard diagram (along with the inevitable "Do Not Erase!" imperative). These days, many architects have abandoned the whiteboard in favor of a tablet attached to an overhead projector. This offers several advantages. First, the tablet has an unlimited canvas and can fit as many drawings as the team might need. Second, it allows the team to copy/paste "what if" scenarios, which would obscure the original if done on a whiteboard. Third, images created on a tablet are already digitized and don't have the inevitable glare of whiteboard photos. Fourth, using electronic images makes remote work much easier and more collaborative.

Eventually, you'll need to create nice diagrams in a fancy tool, but before you invest that time, make sure the team has iterated on the design sufficiently. Whatever platform you're using, you'll find powerful diagramming tools. We happily used OmniGraffle for the original versions of all of the diagrams in this book, which were then refined by O'Reilly illustrators. While we don't necessarily advocate one tool over another, we recommend looking for the following features, as a baseline:

Layers

> Drawing tools often support layers, which architects should learn to use well. Layers allow the user to link groups of items together logically and show or hide them as needed. For example, you might build a comprehensive diagram for a presentation, but hide the overwhelming details when not directly discussing them. Layers also provide a good way to present pictures that build incrementally.

Stencils/templates

> Stencil tools allow users to amass a library of common visual components, including composites of other basic shapes. For example, throughout this book, you have seen standard icons for things like individual microservices; these exist as single items in the authors' stencil tool. Building a set of stencils for patterns and artifacts that are common within an organization creates consistency across architecture diagrams and makes it faster to build new diagrams.

Magnets

> Many drawing tools offer assistance for drawing lines between shapes. *Magnets* represent the places on those shapes where lines snap to connect

automatically, providing alignment and other visual niceties. Some tools allow users to add more magnets or create their own.

Use Layers Semantically, Not Decoratively

We prefer drawing tools that support layers, but we suggest using them *semantically*—that is, in ways that contribute meaning to the overall image. For example, the base layer of each diagram should represent the topology of the architecture: its containers, databases, dependencies, brokers, and other core elements. This layer should focus on architecture rather than implementation; for instance, specify "synchronous communication" rather than naming a specific protocol. The next layer should contain implementation details: what type of database, which communication protocols, and so on.

Using layers in this manner makes diagrams extensible: it's possible to add other contextualized layers to present domain-driven design boundaries, transactional scope, or any other meta-information the architect wants to contrast against the topology of the architecture.

In addition to these specific helpful features, the tool should, of course, support basic functions such as lines, colors, shapes, and other visual artifacts, and should be able to export to a wide variety of formats.

Diagramming Standards: UML, C4, and ArchiMate

The software industry uses several formal standards for technical diagrams. We'll look at three of the most popular here.

UML

Grady Booch, Ivar Jacobson, and Jim Rumbaugh created the Unified Modeling Language (UML) standard in the 1980s to unify their competing design philosophies. It was supposed to be the best of all worlds, but, like many things designed by committee, failed to create much impact outside organizations that mandated its use. Architects and developers still use UML class and sequence diagrams to communicate structure and workflow, but most other UML diagram types have fallen into disuse.

C4

C4 is a diagramming technique developed by Simon Brown between 2006 and 2011 to address UML's deficiencies and modernize its approach. The "four Cs" in C4 are:

Context

> The entire context of the system, including the roles of users and external dependencies.

Container

The physical (and often logical) deployment boundaries and containers within the architecture. This view forms a good meeting point for operations teams and architects.

Component

The component view of the system; this most neatly aligns with an architect's view of the system.

Class

C4 uses the same style of class diagrams as UML, which are effective, so there is no need to replace them.

C4 offers a good alternative for any company seeking to standardize on a diagramming technique. Its creators have been active for years and have gathered a huge following. Critically, it has kept up with changes in the software development ecosystem, expanding to take advantage of new capabilities. Many diagramming tools contain templates for C4 diagrams, and the C4 ecosystem contains tools and frameworks to assist architects. C4 defines standards for components, lines, containers, databases, and other common artifacts.

A C4 diagram of the Silicon Sandwich modular monolith data design appears in Figure 23-2.

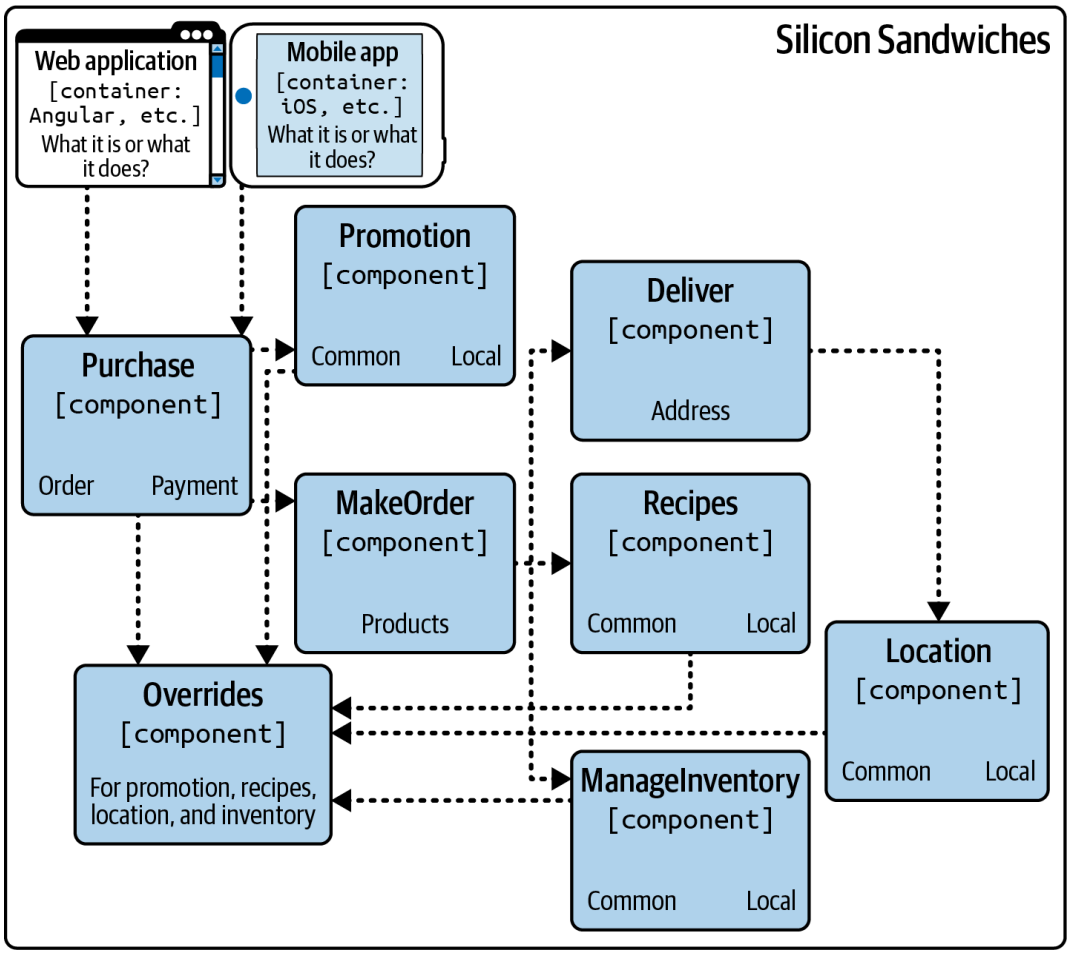

Figure 23-2. Silicon Sandwiches component diagram in C4

ArchiMate

ArchiMate (a portmanteau of *architecture* and *animate*) is an open source enterprise-architecture modeling language from The Open Group for describing,

analyzing, and visualizing architectures within and across business domains. ArchiMate offers a lighter-weight modeling language for enterprise ecosystems. The goal of this technical standard is to be “as small as possible,” not to cover every edge case. It is a popular choice among architects.

Diagram Guidelines

Every architect should build their own diagramming style, whether they use their own modeling language or one of the formal ones. It’s OK to borrow from representations you think are particularly effective. This section offers some general guidelines for creating technical diagrams.

Titles

Make sure all the elements of the diagram have titles, unless they’re very well-known to the audience. Use rotation and other effects to make titles “stick” to the right thing and to make efficient use of space.

Lines

Lines should be thick enough to be clearly visible. If the lines indicate information flow, use arrows to indicate directional or two-way traffic. Different types of arrowheads might suggest different semantics—just be consistent.

One of the few general standards in architecture diagrams is that solid lines almost always indicate synchronous communication, and dotted lines indicate asynchronous communication.

Shapes

While the formal modeling languages we’ve described all have standard shapes, there’s no pervasive set of standard shapes used across the software development world. Most architects make their own set of standard shapes; sometimes these are adopted across an organization to create a standard language.

We tend to use three-dimensional boxes to indicate deployable artifacts, rectangles to indicate containers, and cylinders to represent databases, but we don’t have any particular key beyond that.

Labels

It’s important to label every item in a diagram, especially if there is any chance of ambiguity.

Color

Architects often don’t use color enough. For many years, books were out of necessity printed in black and white, so architects and developers became accustomed to monochromatic drawings. While we still favor monochrome, we use color when it helps distinguish one artifact from another. For example, when discussing microservices architecture quantum in Chapter 19, we use shades of gray in Figure 19-6 (reproduced here as Figure 23-3) to indicate grouping.

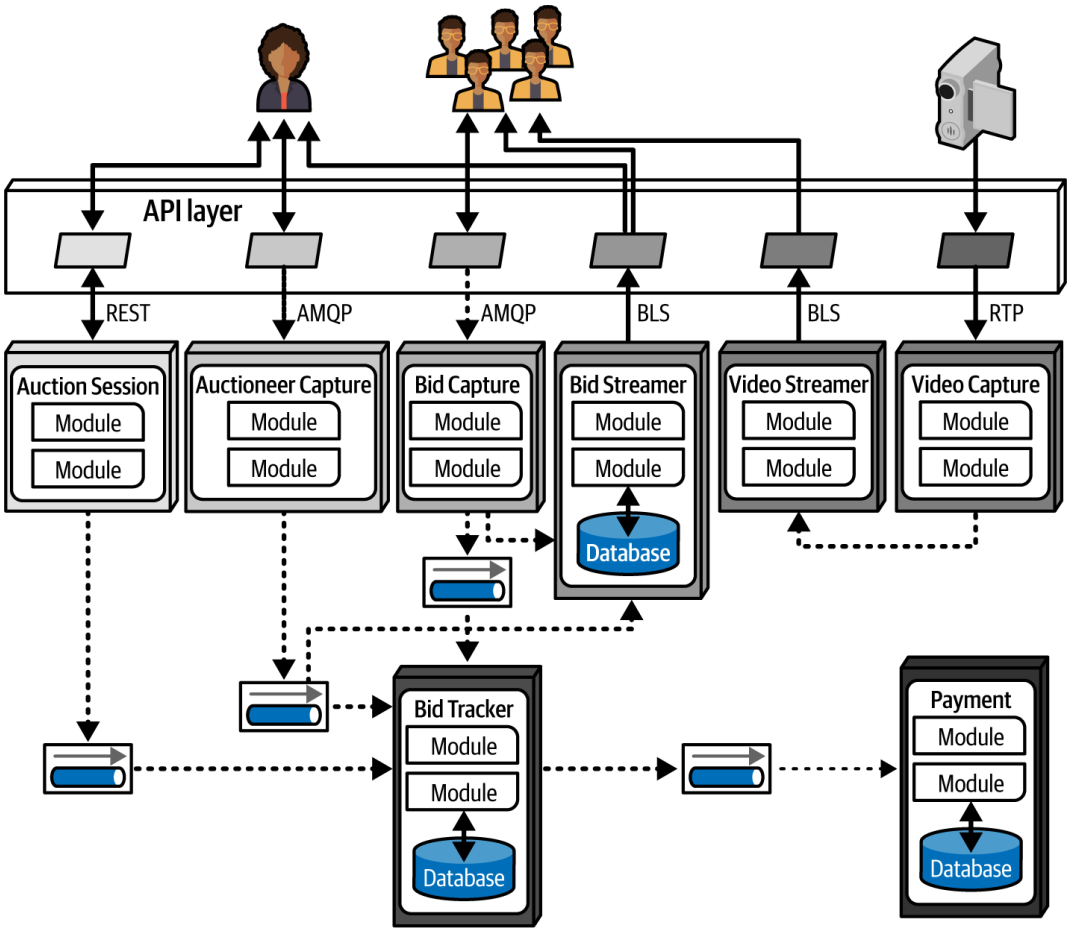

Figure 23-3. Reproduction of a microservices communication example from Chapter 19, showing different services in shades of gray

Be careful about using color to indicate critical differences, however; people who are colorblind or have other visual disabilities will not be able to see the distinction. We suggest using unique iconography in addition to color, so if someone can't distinguish the color, they can still understand the meaning—just as any street-crossing lights use green for go and red for stop, but also show unique figures for each color.

Keys

If a diagram's shapes are ambiguous for any reason, include a key that clearly indicates what each shape represents. An easily misinterpreted diagram is worse than no diagram at all.

Summary

Diagramming standards are a useful way for organizations to provide consistent communication. Architects, however, frequently break the rules, especially when the standard doesn't offer a good way to represent the design. We encourage organizations to establish standards but allow reasonable exceptions.

In the past, when heavyweight computer-assisted software engineering (CASE) tools were the norm, architects had to build elaborate models to represent simple things. They were often forced to include many useless details that were merely noise in that specific context. We favor lightweight diagramming tools and quick-and-dirty artifacts, especially early in the design process. Just don't become so enamored with your creations that you lose your objectivity.