**Software architecture diagramming and patterns**

Graphical user interface, application

Description automatically generated

A software’s architecture is the foundation for any successful software system and will influence everything from maintainability, scalability, stability, and security throughout that system’s lifecycle. The first step toward implementing a new software system is the **architecture diagram**.

As software systems and web applications have become increasingly complex, well-designed system architecture diagrams have become critical for communicating with other developers and stakeholders. Software architecture diagrams are an important documentation practice that will help you plan for and implement changes in your network, visualize strategic initiatives, and stay ahead of your organization’s needs.

Today, we’ll focus on how to diagram, some examples of popular software architecture patterns, and places to find reference architectures to use as a starting point for various use cases. We’ll also go over what a good architectural diagram should accomplish and why you should take the time to create one.

Let’s dive right in!

**We’ll cover**:

* What is software architecture
* Diagramming basics: Flowcharts, C4, and UML 2.5
* 6 software architecture patterns
* 3 public cloud platforms for architecting applications
* Wrapping up and next steps

**What is software architecture?**

Software architecture describes the fundamental concepts and properties of a system within the context of its environment, relationships, principles of design, and more. Software architecture encompasses the organization of a software system, structural elements, behavioral elements, and the composition of those elements into larger subsystems. Software systems can often contain multiple architectures.

“Programming without an overall architecture or design in mind is like exploring a cave with only a flashlight: You don’t know where you’ve been, you don’t know where you’re going, and you don’t know quite where you are.”

Danny Thorpe

Having a great architecture lays the groundwork for how you will deal with performance, fault tolerance, scalability, and reliability in the future. Choosing the right architecture for your software will lead to stable performance under stressful conditions as you scale up.

Even if you don’t anticipate an increase in users, thinking about the big picture for your program and how you can communicate that vision to others can help you and your team makes strategic decisions based on the impact that those decisions will have on your overall architecture.

To extend Danny Thorpe’s cave analogy, having a cave system thoroughly mapped out can make a huge difference if, say, your spelunking trip has gone awry and you’re waiting for rescuers to find you.

Similarly, thorough software architecture can help your engineers quickly locate and fix bugs.

**Do we need to document software architectures?**

Whether you should document your software architecture depends on the type of project you’re working on. There’s some controversy on whether documenting software architectures is actually worth the time or if it slows everything down. The reality is somewhere in the middle.

Some developers don’t need a well-mapped software architecture for every project. If you’re working solo or using inherently adaptable development methodologies that focus on continuous improvements like [Agile](https://www.educative.io/blog/java-agile-development#agile?eid=5082902844932096), it can be difficult to document every change.

“Incorrect documentation is often worse than no documentation.”

Bertrand Meyer

At other times, under-documenting vital parts of software architecture in a complex system can lead to significant issues or contribute to technical debt. As a general rule, as soon as your project expands to include more than one person, you should start documenting a product’s software architecture.

**What do you need to document?**

You must ensure that the core components and high-level architecture are available for every employee. In many cases, the software architecture and its documentation will outlast the initial creators of the product. The people working on the product may change, and the programming language used may change, but the software architecture will almost always remain.

That’s why documenting your software architecture is so important. It allows people working on the project to view a record of how your product changes and evolves.

While you probably don’t want to document every code change or architectural change, you will certainly want to document significant changes and, equally important, why those changes are made.

**What do good architectural diagrams accomplish?**

Good software architecture diagrams are sources of truth and clarity. You want your diagram to quickly convey a software system’s essential composition and behaviors to both technical and non-technical audiences.

High-level architectural diagrams and thorough documentation can be effective tools for communicating the internal state of a system or application when done well.

Diagramming before you start coding and diagramming after your code has been written will also confer different benefits.

* **Forward design** entails creating your diagrams **before** you or your team starts coding. This has the benefit of helping your developers better visualize the system they’re trying to create.
* **Backward design** involves diagramming **after** the code has already been written. This can help developers see how the project developed and document that workflow to improve later.

“There is nothing in the programming field more despicable than an undocumented program.”

Edward Yourdon

You also want your diagram to be as self-explanatory as possible so that anyone looking at it can immediately see the relationships, constraints, and limitations in your software system without needing to ask you what something means.

A good architectural diagram will:

* Diagram core components
* Highlight critical system interactions and relationships
* Be easy to access and share
* Maintain a consistent style
* Make it easier to identify potential flaws or areas for improvement in the software architecture

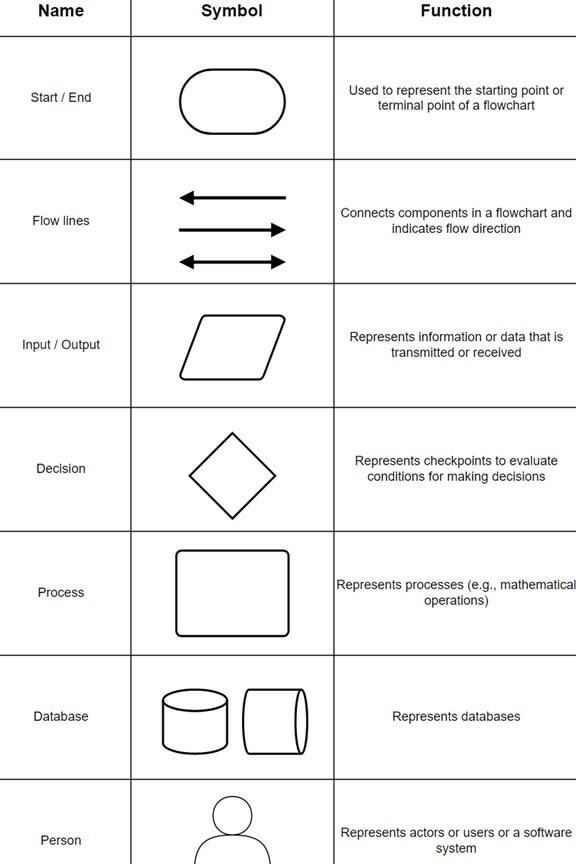
**Diagramming basics: Flowcharts, C4, and UML 2.5**

Now that we’re done talking about how documenting software architecture can benefit you, let’s look at some common methods used to make diagrams.

**Flowcharts**

Flowcharts are one of the most basic types of diagrams you can make. Their simplicity makes them an effective tool for visualizing the logic of an algorithm or program before you start coding.

Here is an example of a flowchart key containing some common symbols used in diagrams.



In a technical diagram, each shape will typically include the following:

* The name of the element being represented
* The role of that element in the system
* The technology of that element

Each component of a diagram will have arrows connecting it to other components and describing the interaction between them.

**The C4 model**

The C4 model is an architectural documentation standard for software systems that breaks a **software system** down into four levels:

* **Context (level 1)**: Context diagrams are high-level, conceptual descriptions of what your system does, what problem it solves, the people involved, and any external systems that interact with it. These diagrams help provide a big picture overview.
* **Containers (level 2)**: Container diagrams go one level deeper to describe high-level interactions between the applications or services that make up your system. Containers can represent APIs, databases, file systems, [microservices](https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started?eid=5082902844932096), etc.
* **Components (level 3)**: Component diagrams look at the different code bodies within a container. These diagrams help visualize abstractions of your codebase.
* **Code (level 4)**: As the name suggests, code diagrams look at architectural elements mapped to code, like classes, interfaces, objects, and functions.

As a **bare minimum**, most teams should create and maintain context and container diagrams for their software system.

Component diagrams can be created if they add value, but you will want to find a way to automate updates to these diagrams for long-term documentation purposes.

Most IDEs (or UML modeling tools) can generate code diagrams on-demand, so documentation at this level of detail is more easily retrieved than maintained. If a component is particularly important or complex, then having level 4 diagrams on hand can be helpful, but for the most part, you can wait until you need these diagrams to generate them.

**UML 2.5**

Unified Modeling Language (UML) is most commonly used in software engineering to create diagrams for documenting level 4 architectural elements.

There are 14 types of UML diagrams falling into two main categories:

* **Structural diagrams** show what objects are within a modeled system.
* **Behavioral diagrams** show how those objects interact with each other.

Table

Description automatically generated with medium confidence

For this article, we’ll be focusing primarily on levels 1 and 2 diagrams of the C4 model, so we won’t get into too much detail here.

However, if you want to generate level 4 diagrams, looking into UML can be a solid place to start.

**Example: Invoicing system**

You are building a web application that digital artists can use to manage and send invoices to clients. What would a context diagram for this look like?

At a minimum, your diagram should include the:

* Database
* Business logic component
* UI

First, start with a representation of your software system.

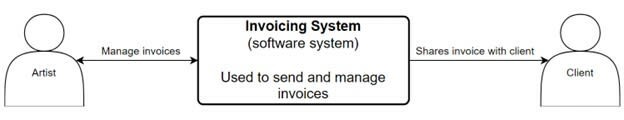
Text

Description automatically generated

Next, you want to document who the actors are. Actors are anyone who will be using the software system.

In this scenario, our actors are:

* Digital artists
* Clients



Now that you know who your actors are, document any **external systems** interacting with the software system.

Diagram

Description automatically generated

If you wanted to go a level deeper, you could create a container diagram. A container diagram would focus on the applications that make up a specific container.

Diagram

Description automatically generated

**6 software architecture patterns**

There are many software architecture styles out there, and being aware of the popular ones can save you some time. Here is a basic (but hopefully comprehensive) look at six different types of architectural patterns.

**1. Layered (N-tier) architecture**

The layered architecture pattern, also known as the N-tier architecture pattern, is the standard architecture used for most Java Enterprise applications. A layered architecture style divides components (or applications) into horizontal, logical layers.

Each layer has a distinct role within the system. One layer may be responsible for handling business logic, while another is responsible for handling presentation logic. This demonstrates a concept known as the separation of concerns (SoC). Components of a layer will **only** deal with logic within that layer. Separating layers like this also makes testing and developing software easier as the pattern itself isn’t too complex. The downside is that this isn’t the most efficient pattern to use and can be difficult to scale up.

Most layered architectures will consist of four closed layers:

* Presentation
* Business
* Persistence
* Database

Occasionally, the business layer and persistence layer are combined into a single layer, especially when the persistence logic (e.g., SQL) is contained within components in the business layer. Smaller applications can have as few as three layers, and more complex applications can contain five or more.

Diagram

Description automatically generated

**Closed** layers require requests to go through layers that precede the target layer. For example, if you are trying to send a request to the database layer, that request **must first** travel through the presentation, business, and persistence layers.

But sometimes, it doesn’t make sense to have a request go through every layer. For situations like this, you can open certain layers so that requests can skip over them and go straight to the layer below them.

Diagram

Description automatically generated

The layered architecture pattern is a great general-purpose pattern for most applications, especially when you’re unsure what kind of architecture pattern to use.

**Note**: **Layers** and **tiers** both refer to functional divisions of a software system. However, a tier refers to software running on infrastructure separated from the other divisions. So, an application with multiple layers could only have one tier if all layers are running on the same device.

**2. Client-server architecture**

In a client-server architecture, there are multiple nodes or clients connected over a network or internet connection who communicate with a central server.

There are two main types of components:

* **Service requesters** (aka clients) that send requests
* **Service providers** that respond to requests

In this architecture, the server hosts, manages and delivers most of the resources and services a client requests. This is also known as a **request-response messaging pattern**.

A couple of classic examples of applications with a client-server architecture are the World Wide Web and email.

Diagram

Description automatically generated

**3. Event-driven architecture**

Event-driven architecture patterns are distributed asynchronous architecture patterns that are highly adaptable. This pattern is best suited for small to large applications with high scalability. Since event-processor components are isolated from each other in this pattern, changes to components can be made without impacting the performance of other components.

There are two main topologies to this pattern: the **mediator** and the **broker** topologies.

Mediator topologies have four main types of components:

* Event queues
* Event mediators
* Event channels
* Event processors

**Mediator topologies** are used when an event has multiple steps that require some level of coordination through a central mediator to be processed.

When a user sends the initial event to an **event queue**, the initial event is then directed to the **event mediator**.

Receiving the initial event prompts the event mediator to publish and send processing events to event channels, telling them to start executing each process step. The event processors receiving the processing events from the event channels contain business logic components that execute all of the steps required to process the initial event.

In general, event-processor components should only perform a single business task without relying on other event processors. This is because you want your event processors to be able to run steps concurrently with other event processors.

Diagram

Description automatically generated

**Broker topologies** are used for process flows where an event does not need a central mediator to distribute or coordinate events.

Broker topologies have two main types of components:

* Brokers
* Event processors

The broker component contains all of the event channels for this event flow. These event channels can be message queues, message topics, or a combination of both.

In the broker topology, event-processor components receive events directly and are responsible for processing and publishing new events to indicate that an event has been processed.

Events **continuously flow** through a chain of processor components until no more events are being published for the initial event.

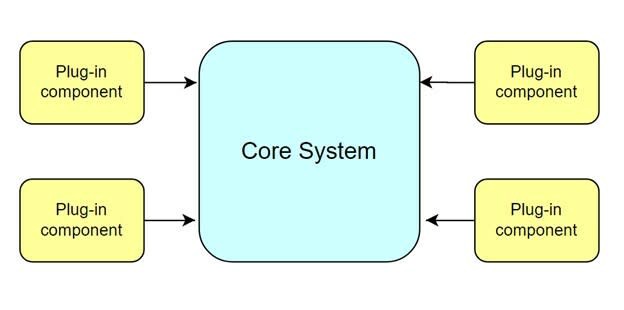
This distribution of processes allows event-driven architectures to run a large number of concurrent connections with minimal resource consumption.

**4. Microkernel architecture**

Microkernel architectures (also known as plug-in architectures) are typically used to implement applications that can be downloaded as a third-party product. This architecture is also commonly found in internal business applications.

One fun thing about this architecture is that you can actually embed it within other patterns, like layered architectures.

There are two types of architectural components in your typical microkernel architecture: a **core system** and **plug-in modules**.



The **core system** contains the minimum business logic needed to make the software system operational. You can extend the software system’s functionality by connecting **plug-in components** to add more features.

It’s kind of like adding a cold-air intake to your car to boost its torque and horsepower.

Plug-in components can be connected using an open service gateway initiative (OSGi), messaging, web services, or object instantiation. The method of implementation is up to you.

**Note**: Plug-in components are independent components meant to extend or enhance the core system’s functionality and **should not** form dependencies with other components.

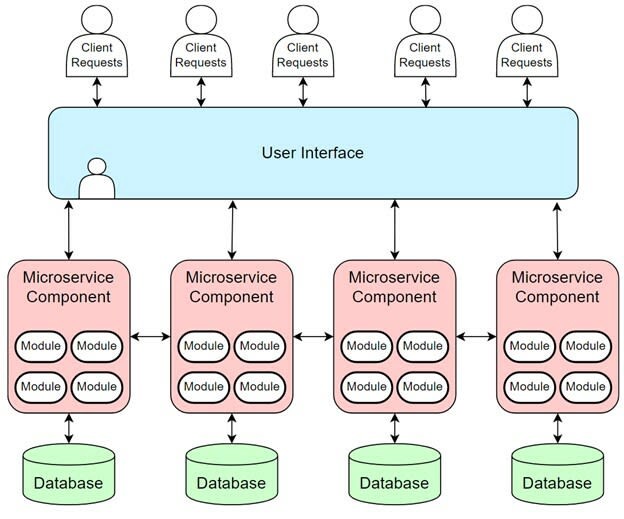
**5. Microservices architecture**

Microservices architectures are one of the most popular software trends at the moment, and one reason for this can be attributed to the easy scalability of development. When microservices can no longer be maintained, they can be rewritten or replaced.

There’s no universally accepted definition for the term **“microservice”**. We will define a microservice as an independently deployable module for this article.

A microservices architecture consists of groups of small, independent, self-contained services with small code bases. Unlike with a monolithic application using a layered architecture pattern, keeping small, separate code bases can minimize the number of dependencies.

Each component of a microservices architecture is deployed as a separate unit. **Separately deploying units** streamlines the delivery pipeline and makes deployment much faster. Development teams can easily build up **continuous delivery** pipelines with smaller units. Testing also becomes easier because you only need to test the features of an individual microservice.



**Note**: Microservices architectures are only effective when deployment is automated because microservices significantly increase the number of deployable units.

Another key concept of the microservices architecture is the **service component**. Service components can range in complexity from single modules to large portions of an application. Microservices architectures are considered a distributed pattern because their service components are **fully decoupled** from one another.

Microservices also facilitate **continuous delivery**, which helps make software development more flexible.

Major companies like Amazon, Netflix, and Spotify can be found implementing this architecture.

**6. Cloud-native architecture**

When you think about typical web applications, most of them typically process requests from clients in the same way. A client sends a request from the web browser, which is sent to the web server, then an application server, and finally, the database server. When this kind of data flow deals with a high volume of concurrently running requests, you typically end up with bottleneck issues. This is where cloud-native architecture patterns come in. Cloud-native patterns are designed to minimize scalability- and concurrency-related issues by removing the central database and using **replicated, in-memory data grids** instead.

The cloud-native architecture is primarily used for [distributed computing systems](https://www.educative.io/blog/distributed-systems-considerations-tradeoffs?eid=5082902844932096) where the interactions between components are mediated through one or more shared spaces.

In this shared space, the components exchange tuples and entries. This brings us to the concept of tuple spaces, or the idea of distributed shared memory.

**Tuple spaces** provide a repository of tuples that can be accessed concurrently. Application data is kept in memory and replicated across active processing units.

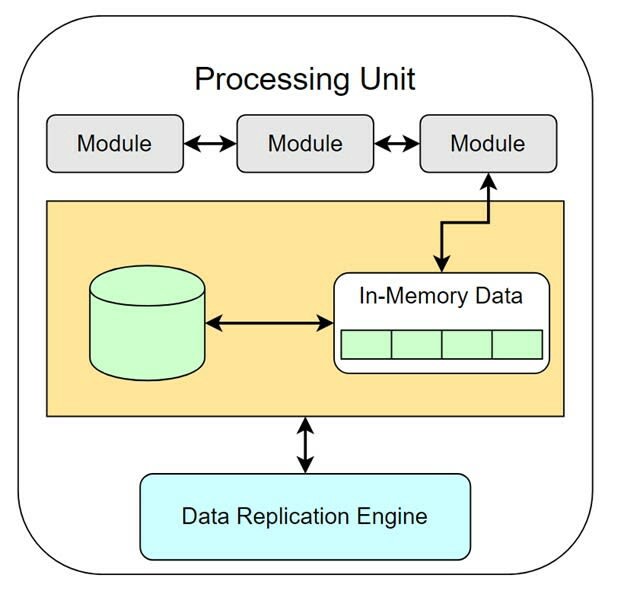
The two main types of components within this architecture pattern are:

* Processing units
* Virtual middleware

Processing units will typically contain:

* Application modules
* An in-memory data grid
* Optional asynchronous persistence store for failover
* A data replication engine

The **data replication engine** is what the **virtual middleware** uses to replicate data changes made in one processing unit across all other active processing units.



The virtualized middleware manages requests, sessions, data replication, distributed request processing, and process-unit deployment.

Diagram

Description automatically generated

The virtualized middleware will contain four main components:

* The messaging grid
* The data grid
* The processing grid
* The deployment manager

The **messaging grid** is the component that manages input requests and session information.

The **data grid** is the most important component in the virtual middleware and interacts with the data replication engines in each processing unit.

The **deployment manager** is the component that manages the startup and shutdown of processing units based on load conditions. It will start up new processing units when user loads increase and shut down processing units when user loads decrease. This dynamic response to changing environments allows space-based applications to scale up easily.

The **processing grid** is an optional component that manages distributed request processing when multiple processing units handle a portion of the application.

This type of software architecture is best for social networking sites or any system that needs to handle massive spikes in traffic.

**3 public cloud platforms for architecting applications**

Reference architectures are great resources for basing your diagrams on. A software reference architecture is like a template of structures and elements arranged to provide a solution for a particular domain or family of software systems.

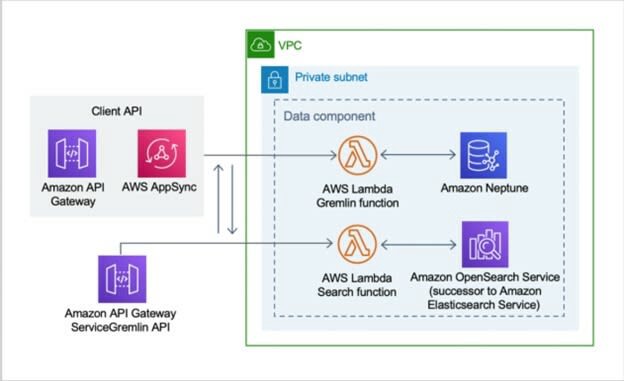
So, if you know that you’re trying to diagram a layered architecture, you can look at existing reference architectures that use that style.

Here are three [public cloud](https://www.educative.io/blog/beginners-guide-cloud-computation?eid=5082902844932096) platforms with free access to hundreds of cloud-based reference architecture diagrams, plus diagramming tools!

**1. Amazon Web Services (AWS)**

[Amazon Web Services](https://www.educative.io/blog/amazon-aws-best-services?eid=5082902844932096) is the most widely used cloud computing platform available today. AWS provides a mix of infrastructure (IaaS), platform (PaaS), and packaged software as a service (SaaS), so it has a wealth of resources like **Workload Discovery**[[2]](https://aws.amazon.com/solutions/implementations/aws-perspective/) and the **AWS CloudFormation Designer**[[2]](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/working-with-templates-cfn-designer.html) for creating, reviewing, and diagramming your application’s architecture.

Here’s an example of an AWS architecture diagram for a data component:



Amazon also provides a [large gallery](https://aws.amazon.com/architecture/reference-architecture-diagrams/?solutions-all.sort-by=item.additionalFields.sortDate&solutions-all.sort-order=desc&whitepapers-main.sort-by=item.additionalFields.sortDate&whitepapers-main.sort-order=desc&awsf.whitepapers-tech-category=*all&awsf.whitepapers-industries=*all) filled with hundreds of real-world architecture diagrams[[3]](https://d1.awsstatic.com/architecture-diagrams/ArchitectureDiagrams/backend-for-frontend-using-api-gateway-ra.pdf?did=wp_card&trk=wp_card). Check it out!

**2. Microsoft Azure**

The Microsoft Azure platform is the second-most popular cloud computing solution and is a great choice if you’re already working with Microsoft products. For hundreds of solutions, you can browse through a [catalog of reference architectures](https://docs.microsoft.com/en-us/azure/architecture/browse/" \t "_blank).

Here’s an example of a reference architecture that describes how a solution uses machine learning to create movie recommendations automatically and at scale.

Timeline

Description automatically generated

Microsoft Azure architectures come with a breakdown of the data flow, an explanation of individual components, and potential use cases. Each architecture also includes a performance analysis based on the five pillars of the [Microsoft Azure Well-Architected Framework](https://docs.microsoft.com/en-us/azure/architecture/framework/):

* **Reliability**: The ability of a system to recover from failures and continue to function
* **Security**: Protection of applications and data from threats
* **Cost optimization**: Management of costs to maximize the value
* **Operational excellence**: Operations processes that keep a system running in production
* **Performance efficiency**: The ability of a system to adapt to changes in load

**3. Google Cloud Platform (GCP)**

The Google Cloud Platform is a collection of cloud computing services and the third-largest IaaS provider. GCP provides a fantastic resource for quickly diagramming:  
[Google Cloud Architecture Diagramming Tool](https://googlecloudcheatsheet.withgoogle.com/architecture).

In the Google Cloud Architecture Diagramming Tool, you can drag and drop prebuilt reference architectures straight onto the canvas. When you’re in the diagramming tool, navigate to the **Diagrams** section and you’ll see over 10 reference architectures for common use cases.

Here’s an example of a reference architecture for a simple, containerized application.

Graphical user interface, diagram, application

Description automatically generated

**Wrapping up and next steps**

Understanding software architecture is an important skill for any software developer to cultivate. Learning how to diagram those patterns effectively can be a great way to help others understand the architecture you’re working on together.

Furthermore, diagramming is a skill you will need to develop if you plan to be a system designer, cloud architect, or solutions architect. Being able to illustrate a clear vision of a product’s software architecture is critical when communicating with non-technical (and technical) audiences. You want to have a clear idea of how different elements of your software system work and pass that on to whoever joins you on a project. If you ever decide to move on from a project, you can rest assured knowing that things won’t fall apart when you leave.

If you enjoyed learning about software architecture patterns and diagramming, we encourage you to keep learning about system design! System design is a great field for people who like to solve abstract problems and think about the bigger picture.

To get started learning these concepts and more, check out Educative’s [Scalability & System Design for Developers](https://www.educative.io/path/scalability-system-design?eid=5082902844932096) learning path.

*Happy learning!*

**Continue reading about system design on Educative**

* [Grokking Modern System Design for Software Engineers and Managers](https://www.educative.io/blog/grokking-modern-system-design?eid=5082902844932096)
* [Scalable Systems 101](https://www.educative.io/blog/scalable-systems-101?eid=5082902844932096)
* [System design fundamentals: What is the CAP theorem?](https://www.educative.io/blog/what-is-cap-theorem?eid=5082902844932096)
* [How to Design a Web Application: Software Architecture 101](https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101?eid=5082902844932096)