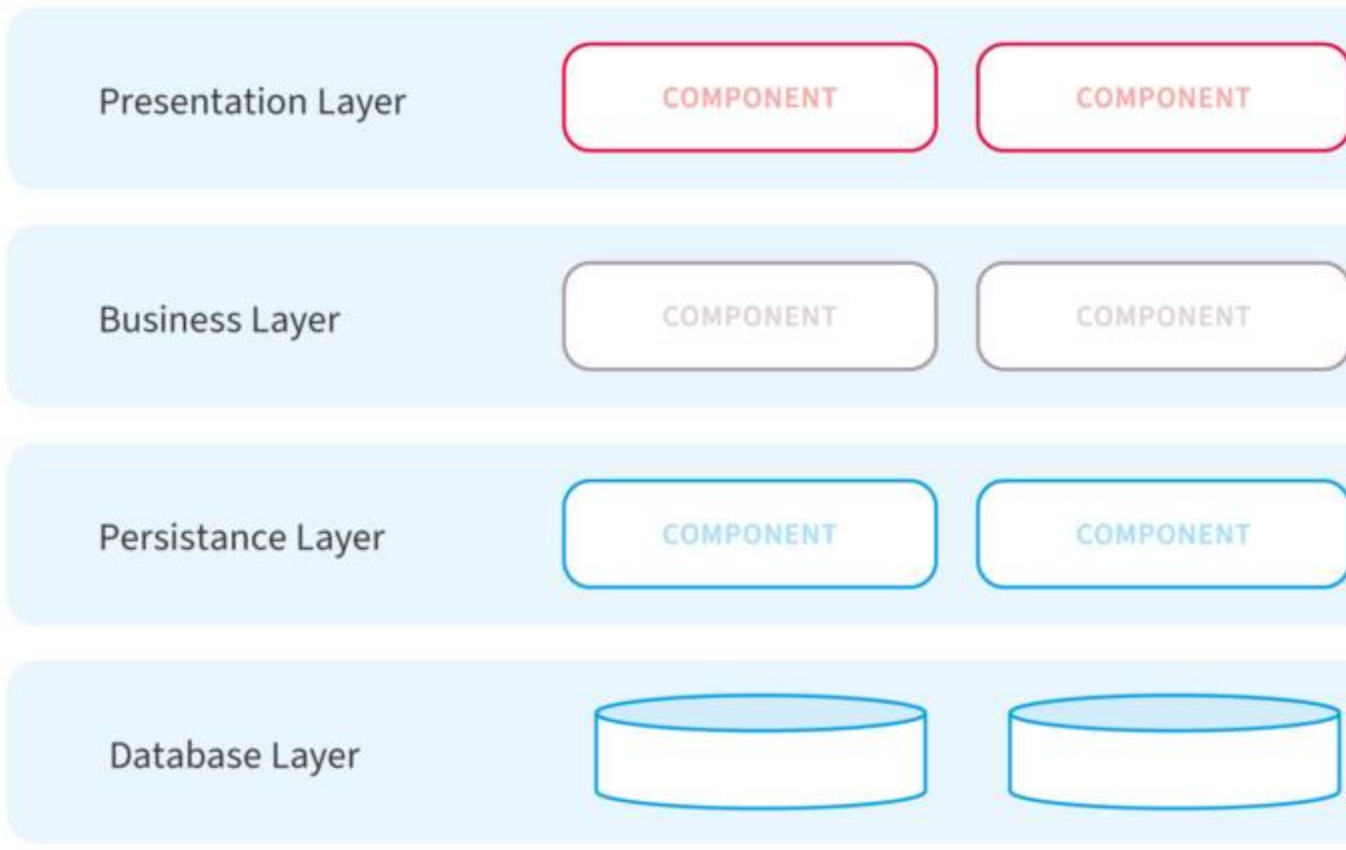
1. Architecture patterns:

10 Common Software Architecture Patterns:

Pattern #1: Layered architecture



Layered pattern

We often use ‘N-tier architecture’, or ‘Multi-tiered architecture’ to denote “layered architecture pattern”. It’s one of the most commonly used patterns where the code is arranged in layers. The key characteristics of this pattern are as follows:

- The outermost layer is where the data enters the system. The data passes through the subsequent layers to reach the innermost layer, which is the database layer.

- Simple implementations of this pattern have at least 3 layers, namely, a presentation layer, an application layer, and a data layer. Users access the presentation layer using a GUI, whereas the application layer runs the business logic. The data layer has a database for the storage and retrieval of data.

This pattern has the following advantages:

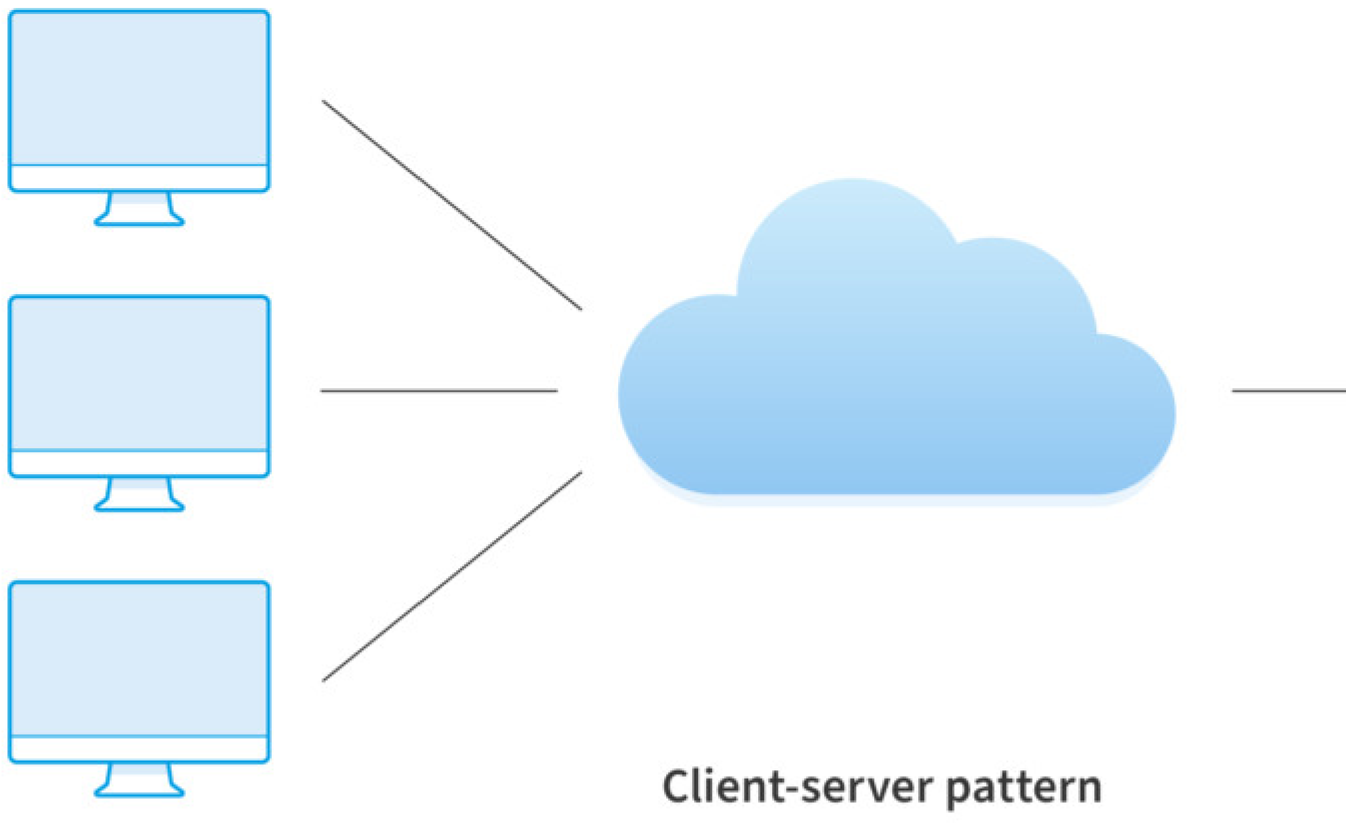
- Maintaining the software is easy since the tiers are segregated.
- Development teams find it easy to manage the software infrastructure, therefore, it's easy to develop large-scale web and cloud-hosted apps.

Popular frameworks like Java EE use this pattern.

There are a few disadvantages too, as follows:

- The code can become too large.
- A considerable part of the code only passes data between layers instead of executing any business logic, which can adversely impact performance.

Pattern #2: Client-server



"Client-server software architecture pattern" is another commonly used one, where there are 2 entities. It has a set of clients and a server. The following are key characteristics of this pattern:

- Client components send requests to the server, which processes them and responds back.
- When a server accepts a request from a client, it opens a connection with the client over a specific protocol.
- Servers can be stateful or stateless. A stateful server can receive multiple requests from clients. It maintains a record of requests from the client, and this record is called a 'session'.

Email applications are good examples of this pattern. The pattern has several advantages, as follows:

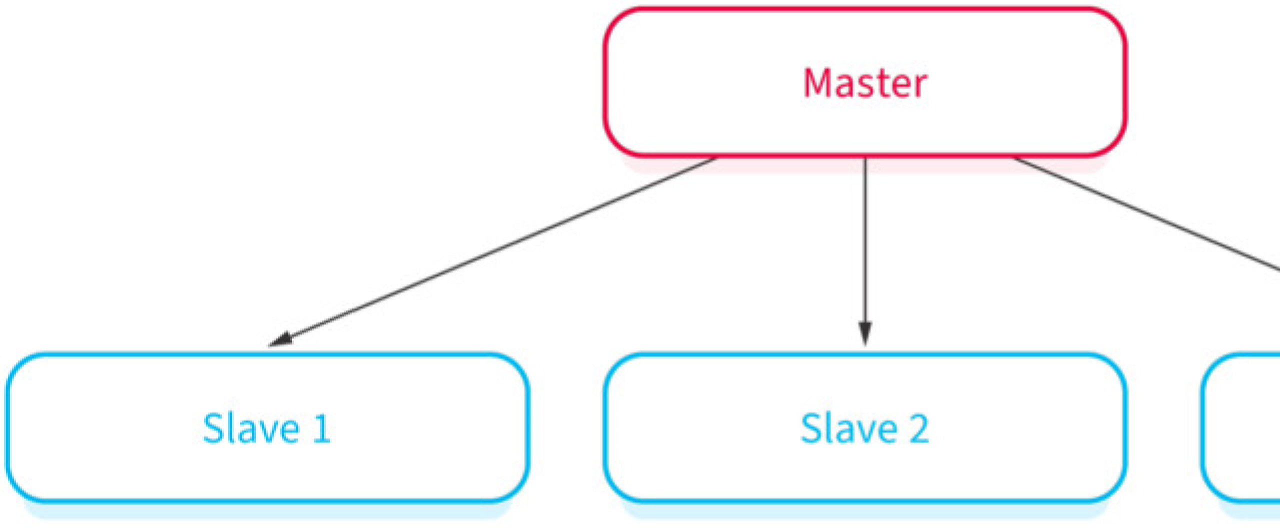
- Clients access data from a server using authorized access, which improves the sharing of data.

- Accessing a service is via a ‘user interface’ (UI), therefore, there’s no need to run terminal sessions or command prompts.
- Client-server applications can be built irrespective of the platform or technology stack.
- This is a distributed model with specific responsibilities for each component, which makes maintenance easier.

Some disadvantages of the client-server architecture are as follows:

- The server can be overloaded when there are too many requests.
- A central server to support multiple clients represents a ‘single point of failure’.

Pattern #3: Master-slave



Master-slave pattern

“Master-slave architecture pattern” is useful when clients make multiple instances of the same request. The requests need simultaneous handling. Following are its’ key characteristics:

- The master launches slaves when it receives simultaneous requests.
- The slaves work in parallel, and the operation is complete only when all slaves complete processing their respective requests.

Advantages of this pattern are the following:

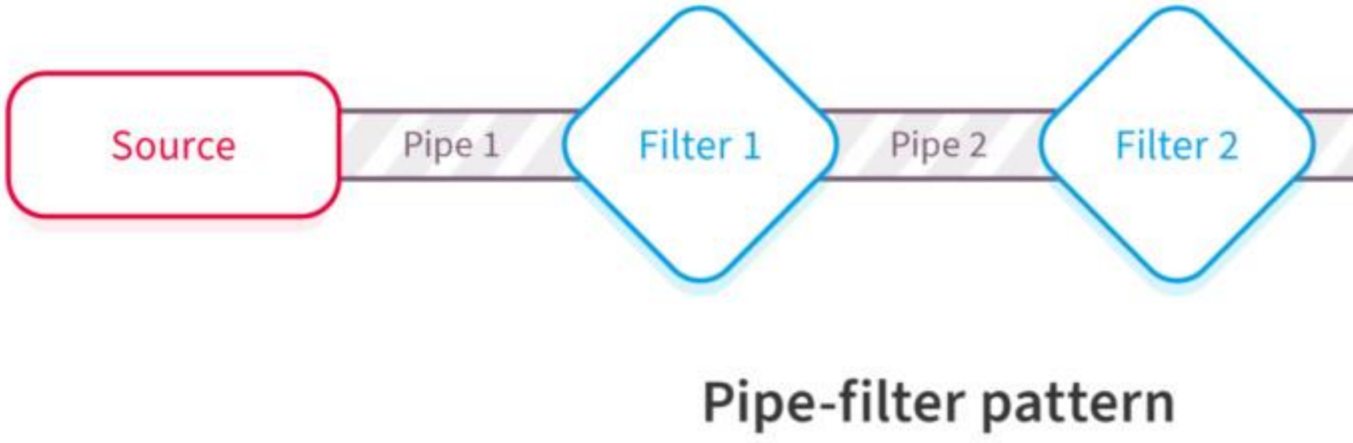
- Applications read from slaves without any impact on the master.
- Taking a slave offline and the later synchronization with the master requires no downtime.

Any application involving multi-threading can make use of this pattern, e.g., monitoring applications used in electrical energy systems.

There are a few disadvantages to this pattern, e.g.:

- This pattern doesn't support automated fail-over systems since a slave needs to be manually promoted to a master if the original master fails.
- Writing data is possible in the master only.
- Failure of a master typically requires downtime and restart, moreover, data loss can happen in such cases.

Pattern #4: Pipe-filter



Suppose you have complex processing in hand. You will likely break it down into separate tasks and process them separately. This is where the "Pipe-filter" architecture pattern comes into use. The following characteristics distinguish it:

- The code for each task is relatively small. You treat it as one independent 'filter'.
- You can deploy, maintain, scale, and reuse code in each filter.
- The stream of data that each filter processes pass through 'pipes'.

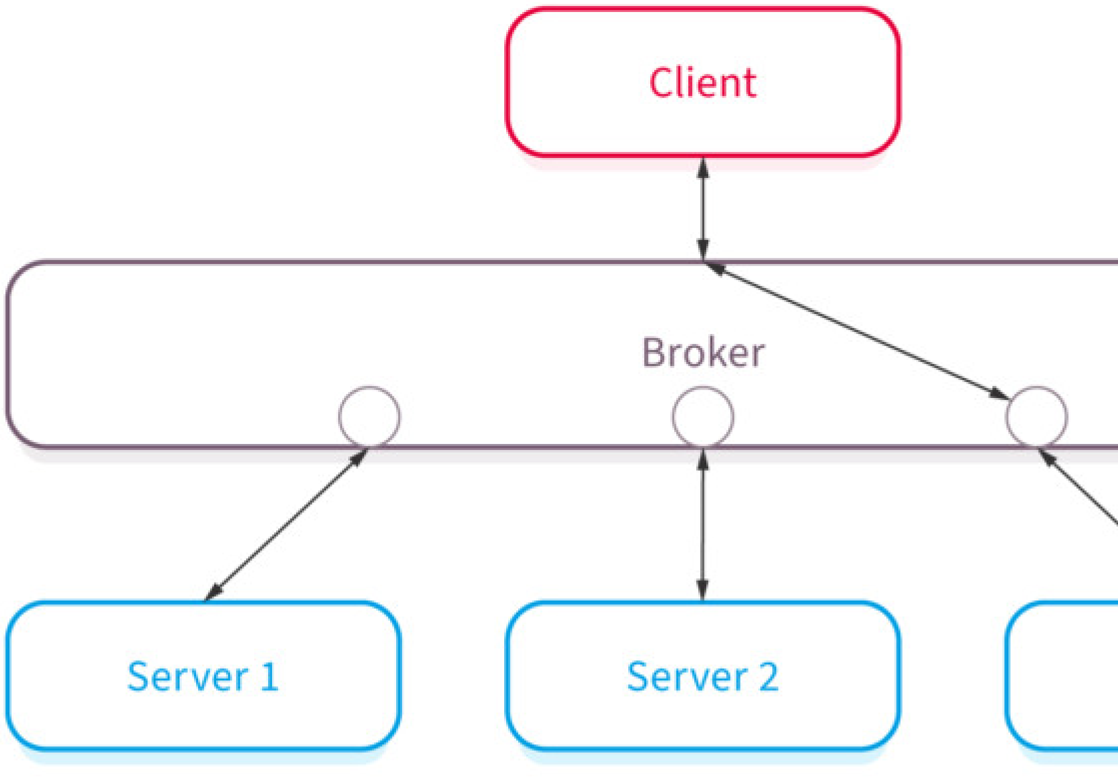
Compilers often use this pattern, due to the following advantages:

- There are repetitive steps such as reading the source code, parsing, generating code, etc. These can be easily organized as separate filters.
- Each filter can perform its' processing in parallel if the data input is arranged as streams using pipes.
- It's a resilient model since the pipeline can reschedule the work and assign to another instance of that filter.

Watch out for a few disadvantages:

- This pattern is complex.
- Data loss between filters is possible in case of failures unless you use a reliable infrastructure.

Pattern #5: Broker



Broker pattern

Consider distributed systems with components that provide different services independent of each other. Independent components could be heterogeneous systems on different servers, however, clients still need their requests serviced. “Broker architecture pattern” is a solution to this.

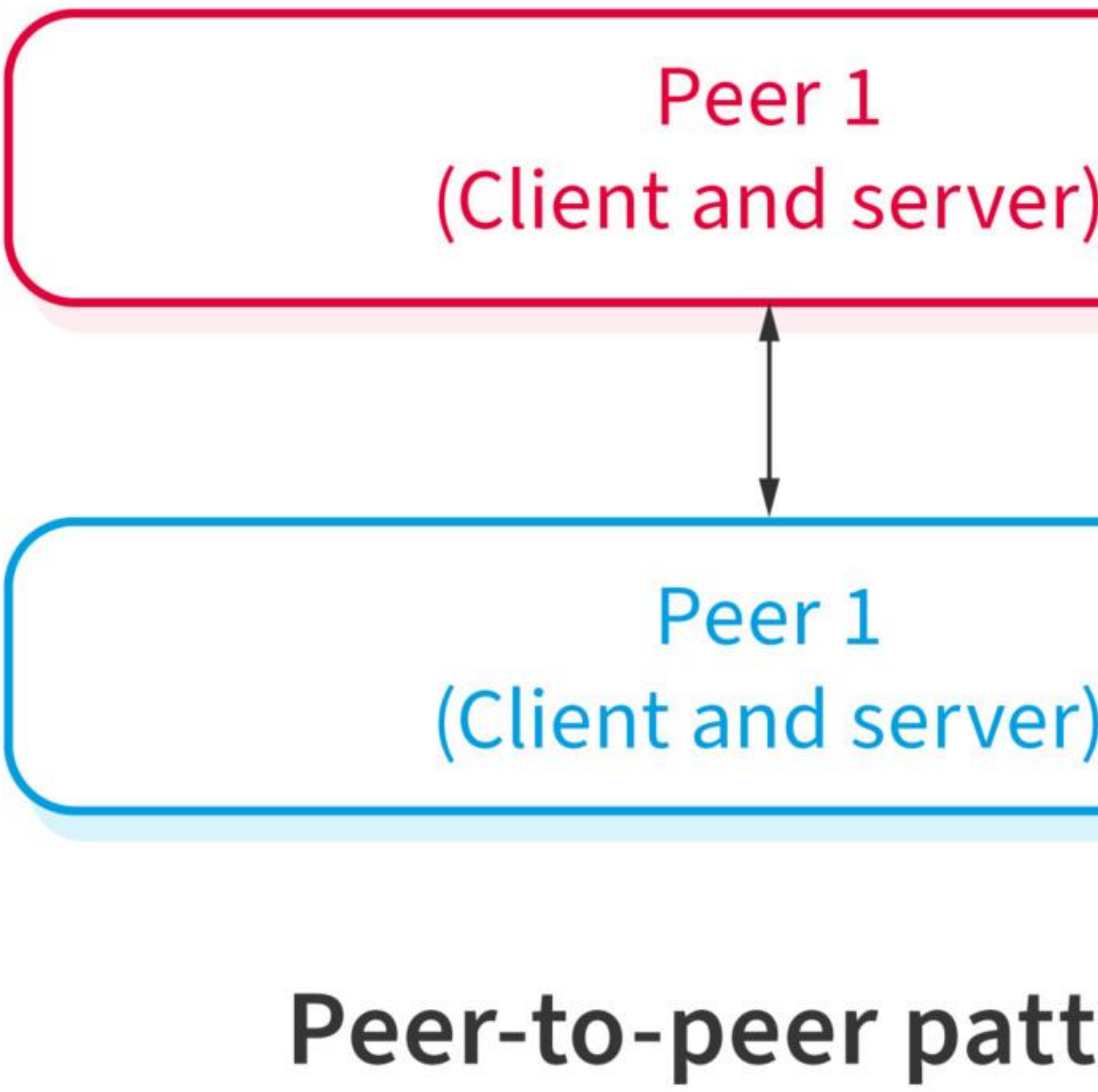
It has the following broad characteristics:

- A broker component coordinates requests and responses between clients and servers.
- The broker has the details of the servers and the individual services they provide.
- The main components of the broker architectural pattern are clients, servers, and brokers. It also has bridges and proxies for clients and servers.
- Clients send requests, and the broker finds the right server to route the request to.
- It also sends the responses back to the clients.

Message broker software like IBM MQ uses this pattern. The pattern has a few distinct advantages, e.g.:

- Developers face no constraints due to the distributed environment, they simply use a broker.
- This pattern helps using object-oriented technology in a distributed environment.

Pattern #6: Peer-to-peer (P2P)



“Peer-to-peer (P2P) pattern” is markedly different from the client-server pattern since each computer on the network has the same authority. Key characteristics of the P2P pattern are as follows:

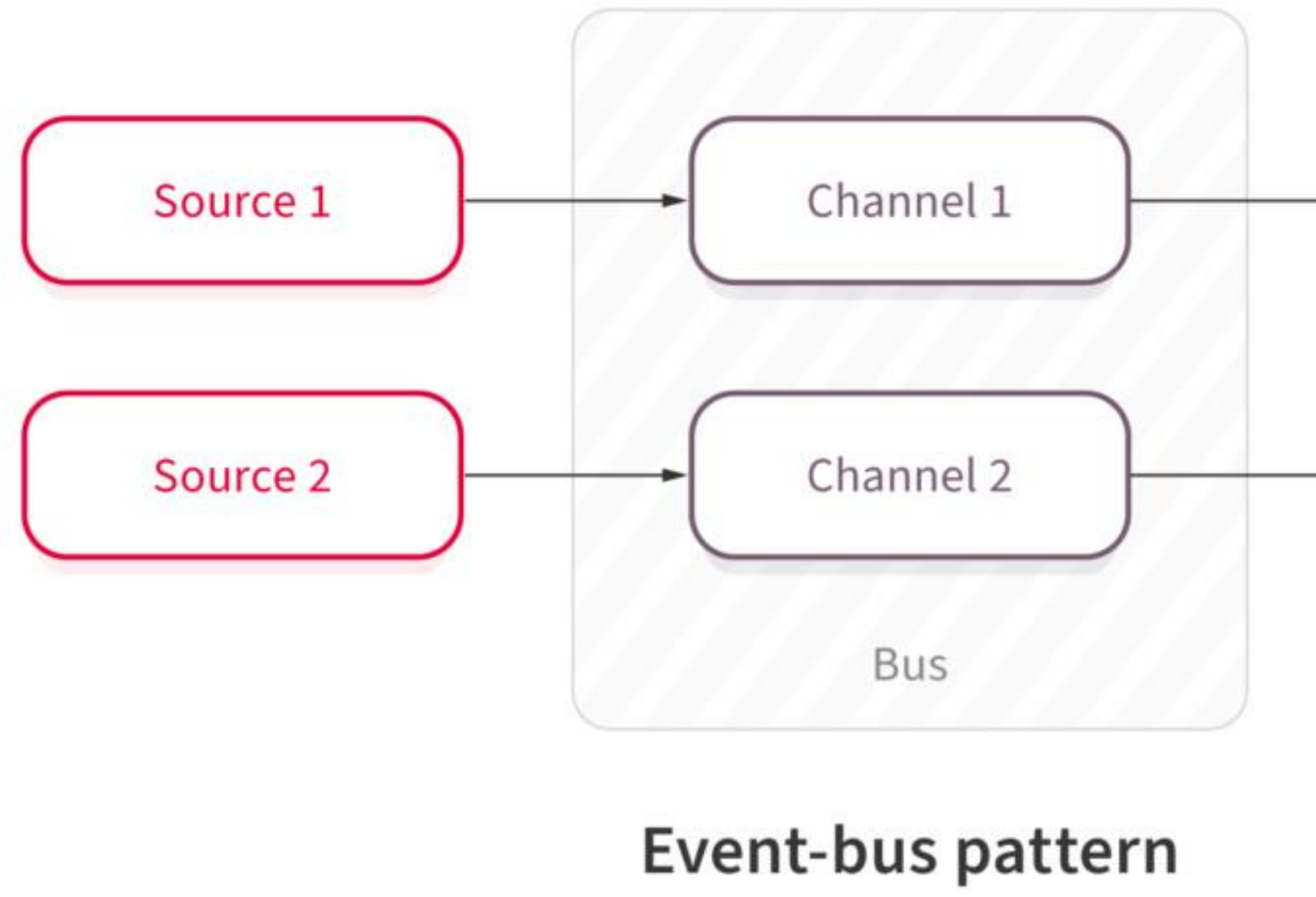
- There isn’t one central server, with each node having equal capabilities.
- Each computer can function as a client or a server.
- When more computers join the network, the overall capacity of the network increases.

File-sharing networks are good examples of the P2P pattern. Bitcoin and other cryptocurrency networks are other examples. The advantages of a P2P network are as follows:

- P2P networks are decentralized, therefore, they are more secure. You must have already heard a lot about the security of the Bitcoin network.
- Hackers can’t destroy the network by compromising just one server.

Under heavy load, the P2P pattern has performance limitations, as the questions surrounding the Bitcoin transaction throughout show.

Pattern #7: Event-bus pattern



There are applications when components act only when there is data to be processed. At other times, these components are inactive. “Event-bus pattern” works well for these, and it has the following characteristics:

- A central agent, which is an event-bus, accepts the input.
- Different components handle different functions, therefore, the event-bus routes the data to the appropriate module.
- Modules that don’t receive any data pertaining to their function will remain inactive.

Think of a website using JavaScript. Users’ mouse clicks and keystrokes are the data inputs. The event-bus will collate these inputs and it will send the data to appropriate modules. The advantages of this pattern are as follows:

- This pattern helps developers handle complexity.
- It’s a scalable architecture pattern.

- This is an extensible architecture, new functionalities will only require a new type of events.

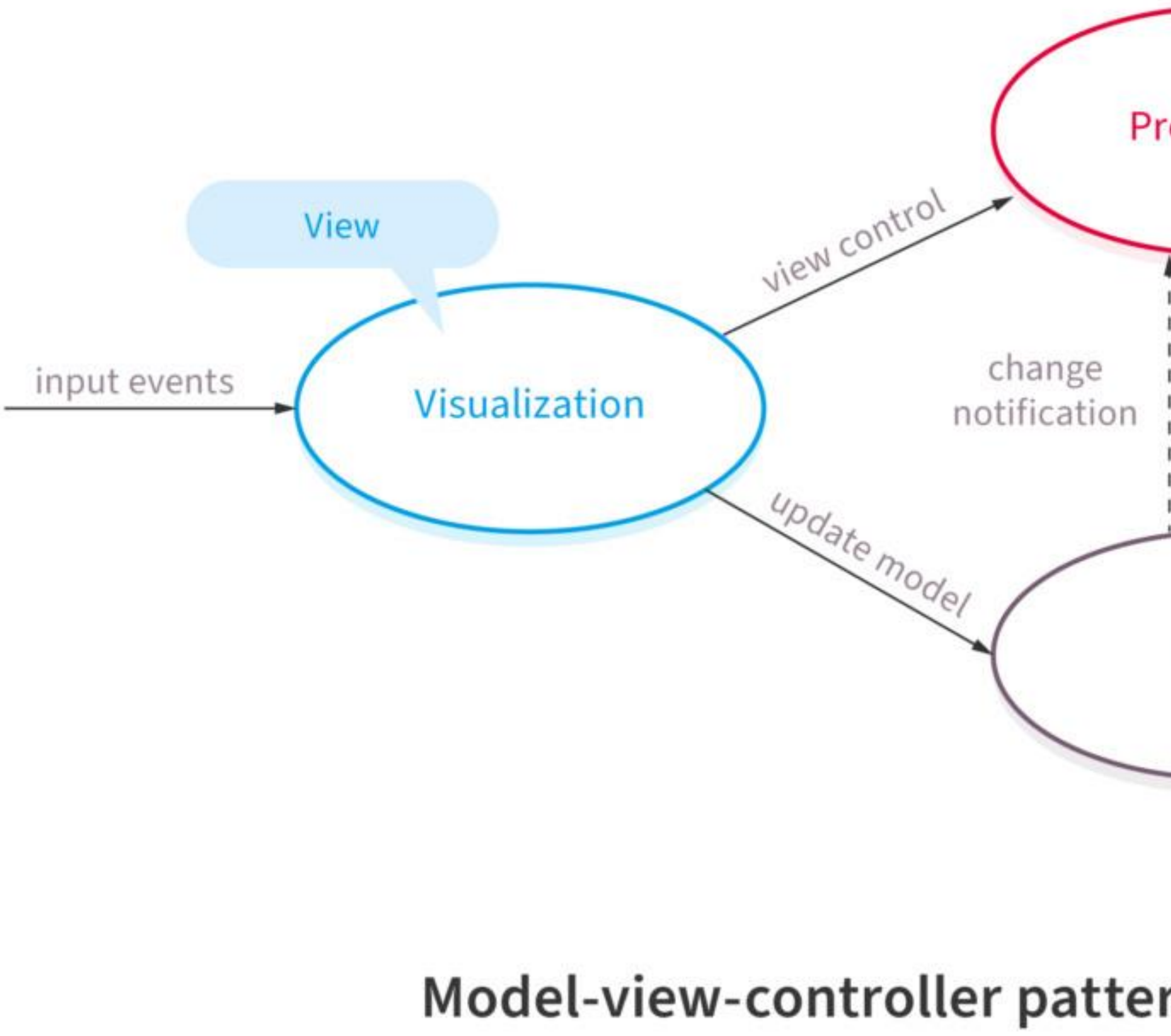
This software architecture pattern is also used in Android development.

Some disadvantages of this pattern are as follows:

- Testing of interdependent components is an elaborate process.
- If different components handle the same event require complex treatment to error-handling.
- Some amount of messaging overhead is typical of this pattern.

The development team should make provision for sufficient fall-back options in the event the event-bus has a failure.

Pattern #8: Model-View-Controller (MVC)



“Model-View-Controller (MVC) architecture pattern” involves separating an applications’ data model, presentation layer, and control aspects. Following are its’ characteristics:

- There are three building blocks here, namely, model, view, and controller.
- The application data resides in the model.
- Users see the application data through the view, however, the view can’t influence what the user will do with the data.

- The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action is typically a method call to the model. The response is shown in the view.

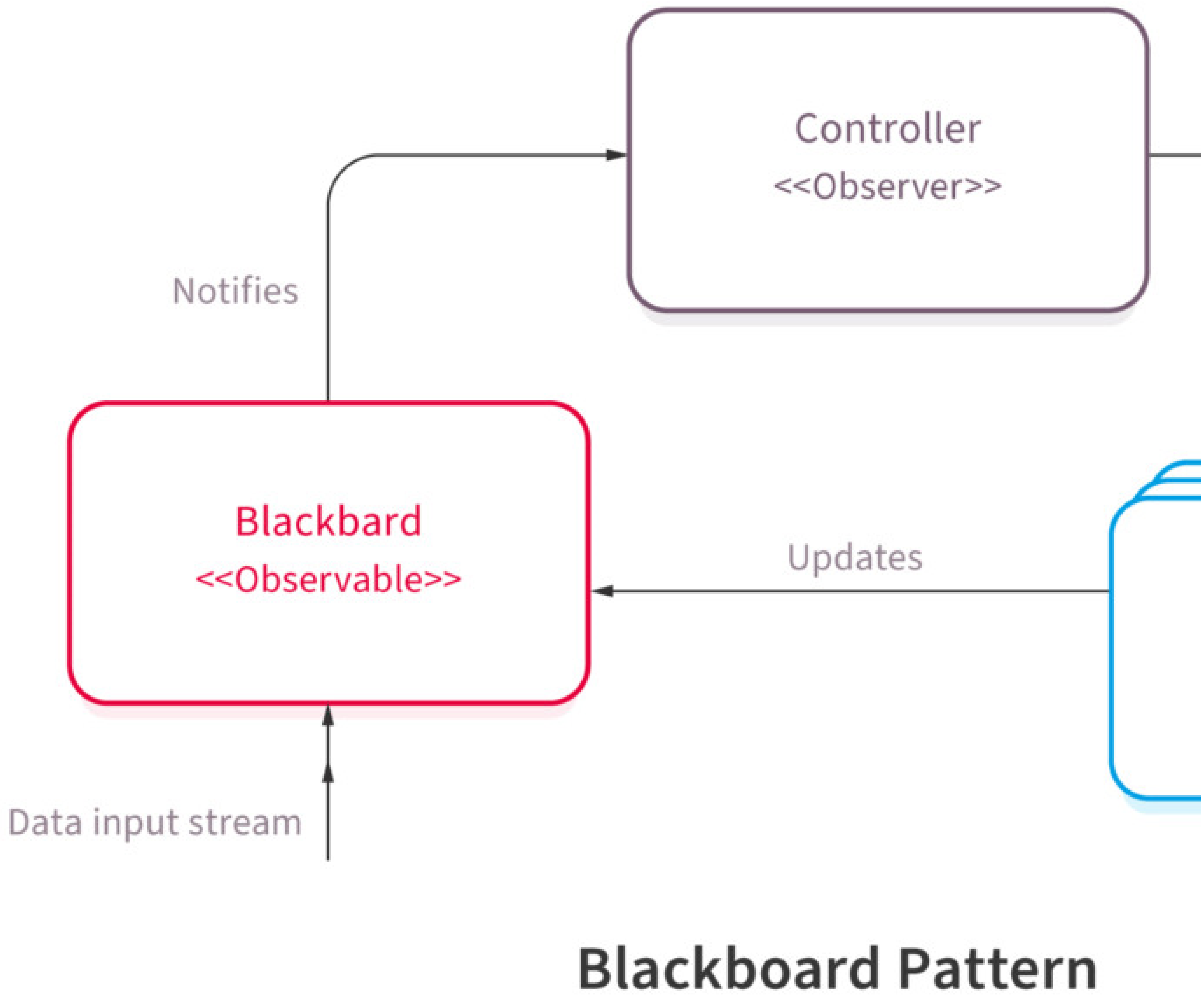
This pattern is popular. Many web frameworks like Spring and Rails use it, therefore, many web applications utilize this pattern. Its' advantages are as follows:

- Using this model expedites the development.
- Development teams can present multiple views to users.
- Changes to the UI is common in web applications, however, the MVC pattern doesn't need changes for it.
- The model doesn't format data before presenting to users, therefore, you can use this pattern with any interface.

There are also a few disadvantages, for e.g.:

- With this pattern, the code has new layers, making it harder to navigate the code.
- There is typically a learning curve for this pattern, and developers need to know multiple technologies.

Pattern #9: Blackboard



Emerging from the world of ‘Artificial Intelligence’ (AI) development, the “Blackboard architecture pattern” is more of a stop-gap arrangement. Its’ noticeable characteristics are as follows:

- When you deal with an emerging domain like AI or ‘Machine Learning’ (ML), you don’t necessarily have a settled architecture pattern to use. You start with the blackboard pattern, subsequently, when the domain matures, you adopt a different architecture pattern.
- There are three components, namely, the blackboard, a collection of knowledge resources, and a controller.

- The application system stores the relevant information in the blackboard.
- The knowledge resources could be algorithms in the AI or ML context that collect information and updates the blackboard.
- The controller reads from the blackboard and updates the application ‘assets’, for e.g., robots.

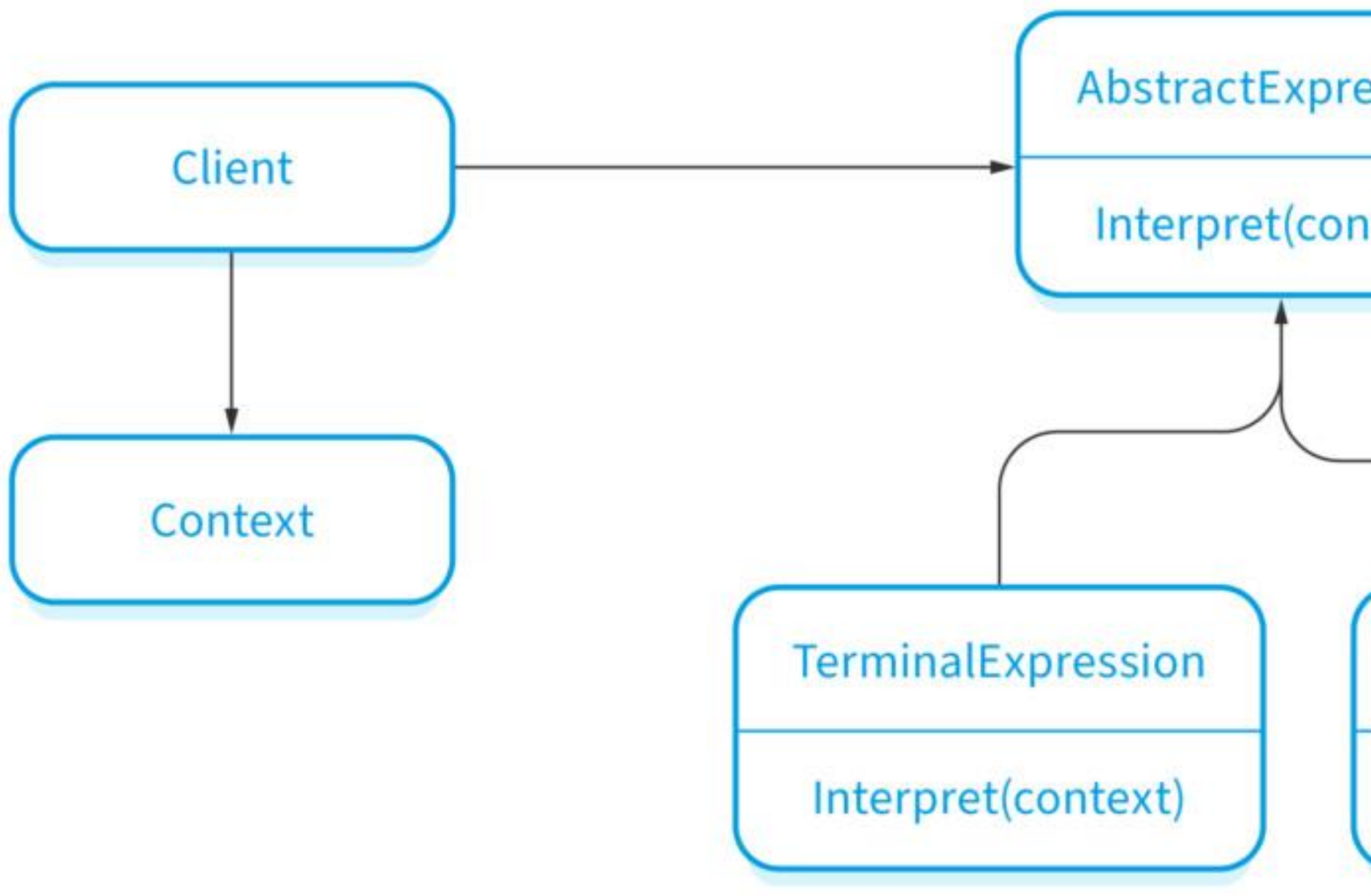
Image recognition, speech recognition, etc. use this architecture pattern. It has a few advantages, as follows:

- The pattern facilitates experiments.
- You can reuse the knowledge resources like algorithms.

There are also limitations, for e.g.:

- It’s an intermediate arrangement. Ultimately, you will need to arrive at a suitable architecture pattern, however, you don’t have certainty that you will find the right answer.
- All communication within the system happens via the blackboard, therefore, the application can’t handle parallel processing.
- Testing can be hard.

Pattern #10: Interpreter



Interpreter Pattern

A pattern specific to certain use cases, the "Interpreter pattern" deals with the grammar of programming languages. It offers an interpreter for the language. It works as follows:

- You implement an interface that aids in interpreting given contexts in a programming language.
- The pattern uses a hierarchy of expressions.
- It also uses a tree structure, which contains the expressions.
- A parser, external to the pattern, generates the tree structure for evaluating the expressions.

The use of this pattern is in creating “Classes” from symbols in programming languages. You create a grammar for the language, so that interpretation of sentences becomes possible. Network protocol languages and SQL uses this pattern.

2. What Is a Cron Job?

Cron is a utility program that lets users input commands for scheduling tasks repeatedly at a specific time. Tasks scheduled in cron are called **cron jobs**. Users can determine what kind of task they want to automate and when it should be executed.

Cron is a **daemon** – a background process executing non-interactive jobs. In Windows, you might be familiar with background processes such as **Services** that work similarly to the cron daemon.

A daemon is always idle, waiting for a command to request it to perform a particular task. The command can be input on any computer on the network.

A cron file is a simple text file that contains commands to run periodically at a specific time. The default system cron table or crontab configuration file is **/etc/crontab**, located within the crontab directory **/etc/cron.*/**.

Only system administrators can edit the system crontab file. However, Unix-like operating systems support multiple admins. Each can create a crontab file and write commands to perform jobs anytime they want.

With cron jobs, users can automate system maintenance, disk space monitoring, and schedule backups. Because of their nature, cron jobs are great for computers that work 24/7, such as servers.

While cron jobs are used mainly by system administrators, they can be beneficial for web developers too.

For instance, as a website administrator, you can set up one cron job to automatically backup your site every day at midnight, another to check for broken links every Monday at midnight, and a third to clear your site cache every Friday at noon.

However, like any other program, cron has limitations you should consider before using it:

- **The shortest interval between jobs is 60 seconds**. With cron, you won't be able to repeat a job every 59 seconds or less.
- **Centralized on one computer.** Cron jobs can't be distributed to multiple computers on a network. So if the computer running cron crashes, the scheduled tasks won't be executed, and the missed jobs will only be able to be run manually.
- **No auto-retry mechanism**. Cron is designed to run at strictly specified times. If a task fails, it won't run again until the next scheduled time. This makes cron unsuitable for incremental tasks.

With these limitations, cron is an excellent solution for simple tasks that run at a specific time with regular intervals of at least 60 seconds.

If you want to schedule a one-time job for later, you might want to use **another method**.

Pro Tip

Before creating a Cron Job, make sure that your script works. To do that, open the file in your browser (by URL) or execute it via SSH, depending on what type of script you have. If your script doesn't work, contact developers for help.

Basic Cron Job Operations

This tutorial will show you how to schedule cron jobs by inputting **commands** into a shell program like Bash on Linux or another Unix-like operating system.

Hostinger's **VPS hosting** runs on a Linux-based operating system. Therefore, learning how to schedule cron jobs will significantly increase your work efficiency as a VPS administrator. The command line for VPS can be accessed through **PuTTY SSH**.

Before proceeding with the basic operations of cron, it's essential to know the different cron job configuration files:

- **The system crontab.** Use it to schedule system-wide, essential jobs that can only be changed with root privileges.

- **The user crontab.** This file lets users create and edit cron jobs that only apply at the user level.

If you want to edit the system crontab, make sure that the current user has root privileges.

The following are some basic operations that cron can perform:

To create or edit a crontab file, enter the following into the command line:

```
crontab -e
```

If no crontab files are found in your system, the command will automatically create a new one. **crontab -e** allows you to add, edit, and delete cron jobs.

You'll need a text editor like **vi** or **nano** to edit a crontab file. When entering **crontab -e** for the first time, you'll be asked to choose which text editor you want to edit the file with.

To see a list of active scheduled tasks in your system, enter the following command:

```
crontab -l
```

If your system has multiple users, you can **view their crontab file lists** by entering the following command as a superuser:

```
crontab -u username -l
```

You can also easily **edit other users' scheduled jobs** by typing the following crontab command:

```
sudo su crontab -u username -e
```

To give yourself root privileges, append **sudo su** to the beginning of the command. Some commands, including this one, can only be executed by root users.

Lastly, **to delete all scheduled tasks** in your crontab file and start fresh, type the following command:

```
crontab -r
```

Alternatively, the following command is the same as **crontab -r**, except it will prompt the user with a yes/no option before removing the crontab:

```
crontab -i
```

In addition to crontab, the root user can also add cron jobs to the **etc/cron.d** directory. It's most suitable for running scripts for automatic installations and updates.

Keep in mind that the user adding cron jobs to this directory must have root access and conform to **run-parts**' naming conventions.

Alternatively, a root user can move their scripts into the following directories to schedule their execution:

- **/etc/cron.hourly/** – Run all scripts once an hour
- **/etc/cron.daily/** – Run once a day.
- **/etc/cron.weekly/** – Run once a week.
- **/etc/cron.monthly/** – Run once a month.

Crontab Syntax

To create a cron job, you'll need to understand cron's syntax and formatting first. Otherwise, correctly setting up cron jobs may not be possible.

The crontab syntax consists of five fields with the following possible values:

- **Minute.** The minute of the hour the command will run on, ranging from 0-59.
- **Hour.** The hour the command will run at, ranging from 0-23 in the 24-hour notation.
- **Day of the month**. The day of the month the user wants the command to run on, ranging from 1-31.
- **Month**. The month that the user wants the command to run in, ranging from 1-12, thus representing January-December.
- **Day of the week.** The day of the week for a command to run on, ranging from 0-6, representing Sunday-Saturday. In some systems, the value 7 represents Sunday.

Don't leave any of the fields blank.

If, for example, you want to set up a cron job to run **root/backup.sh** every Friday at 5:37 pm, here's what your cron command should look like:

```
37 17 * * 5 root/backup.sh
```

In the example above, **37** and **17** represent 5:37 pm. Both asterisks for the **Day of the month** and **Month** fields signify all possible values. This means that the task should be repeated no matter the date or the month. Finally, **5** represents Friday. The set of numbers is then followed by the location of the task itself.

If you're not sure about manually writing the cron syntax, you can use free tools like **Crontab Generator** or **Crontab.guru** to generate the exact numbers for the time and date you want for your command.

To set the correct time for your cron command, knowledge of cron job operators is essential. They allow you to specify which values you want to enter in each field. You need to use proper operators in all crontab files.

- **Asterisk (*)**. Use this operator to signify all possible values in a field. For example, if you want your cron job to run every minute, write an asterisk in the **Minute** field.
- **Comma (,)**. Use this operator to list multiple values. For example, writing **1,5** in the **Day of the week** field will schedule the task to be performed every Monday and Friday.
- **Hyphen (-)**. Use this operator to determine a range of values. For example, if you want to set up a cron job from June to September, writing **6-9** in the **Month** field will do the job.
- **Separator (/)**. Use this operator to divide a value. For example, if you want to make a script run every twelve hours, write ***/12** in the **Hour** field.
- **Last (L)**. This operator can be used in the day-of-month and day-of-week fields. For example, writing **3L** in the day-of-week field means the last Wednesday of a month.
- **Weekday (W)**. Use this operator to determine the closest weekday from a given time. For example, if the **1st** of a month is a Saturday, writing **1W** in the day-of-month field will run the command on the following Monday (the **3rd**).
- **Hash (#).** Use this operator to determine the day of the week, followed by a number ranging from 1 to 5. For example, **1#2** means the second Monday of the month.
- **Question mark (?)**. Use this operator to input "no specific value" for the "day of the month" and "day of the week" fields.

Cron Job Special Strings

Special strings are used to schedule cron jobs at time intervals without the user having to figure out the logical set of numbers to input. To use them, write an **@** followed by a simple phrase.

Here are some useful special strings that you can use in commands:

- **@hourly**. The job will run once an hour.
- **@daily** or **@midnight**. These strings will run the task every day at midnight.
- **@weekly**. Use this to run jobs once a week at midnight on Sunday.
- **@monthly.** This special string runs a command once on the first day of every month.
- **@yearly**. Use this to run a task once a year at midnight on January 1st.
- **@reboot**. With this string, the job will run only once at startup.

Important! Remember to take extra care when scheduling timezone-sensitive cron jobs.

Cron Syntax Examples

Now that you know how correct cron syntax looks, we'll go over some examples to help you understand it better.

Keep in mind that the cron output will be automatically sent to your local email account. If you want to stop receiving emails, you can add **>/dev/null 2>&1** to a command as in the following example:

```
0 5 * * * /root/backup.sh >/dev/null 2>&1
```

If you want to send the output to a specific email account, add **MAILTO** followed by an email address. Here is an example:

```
MAILTO="myname@hostinger.com"
0 3 * * * /root/backup.sh >/dev/null 2>&1
```

Take a look at the following sample commands to get a better understanding of the cron syntax:

Example	Explanation
0 0 * * 0 /root/backup.sh	Perform a backup every Sunday at midnight.
0 * * * 1 /root/clearcache.sh	Clear the cache every hour on Mondays.
0 6,18 * * * /root/backup.sh	Backup data twice a day at 6am and 6pm.
***/10 * * * * /scripts/monitor.sh**	Perform monitoring every 10 minutes.
***/15 * * * * /root/backup.sh**	Perform a backup every 15 minutes.
*** * 20 7 * /root/backup.sh**	Perform a backup every minute on July 20.
0 22 * * 1-5 /root/clearcache.sh	Clear the cache every weekday (Monday to Friday) at 10pm.
0 0 * * 2 * /root/backup.sh	Perform a backup at midnight every Tuesday.
*** * * 1,2,5 * /scripts/monitor.sh**	Perform monitoring every minute during January, February, and May.
10-59/10 5 * * * /root/clearcache.sh	Clear the cache every 10 minutes at 5am, starting from 5:10am.
0 8 1 */3 * /home/user/script.sh	Make the task run quarterly on the first day of the month at 8am.
0 * * * * /root/backup.sh	Create a backup every hour.
*** * * * * /scripts/script.sh; /scripts/scrit2.sh**	Include multiple tasks on a single cron job. Useful for scheduling multiple tasks to run at the same time.
@reboot /root/clearcache.sh	Clear cache every time you turn on the system.
0 8 1-7 * 1 /scripts/script.sh	Run a script on the first Monday of each month, at 8 am.
5 4 * * 0 /root/backup.sh	Create a backup every Sunday at 4:05 am.
15 9 1,20 * * /scripts/monitor.sh	Perform monitoring at 9:15 pm on the 1st and 20th of every month.
@hourly /scripts/monitor.sh	Perform monitoring every hour.
0 0 1,15 * 3 /scripts/script.sh	Run a script at midnight every Wednesday between the 1st and 15th of every month.

15 14 1 * * /root/clearcache.sh	Clear the cache on the first day of every month at 2:15 pm.
00 08-17 * * 1-5 bin/check-db-status	Check database status every day from Monday to Friday every hour from 8 to 5 pm.
15 6 1 1 * /root/backup.sh	Perform a backup every January 1st at 6:15am.
0 0 * * * /scripts/monitor.sh	Run the monitoring script once a day at midnight.
0 0 15 * * /root/clearcache.sh	Clear the cache at midnight on the 15th of every month.
*** * * * 1,5 /scripts/monitor.sh**	Performing monitoring every Monday and Friday.

Cron Permissions

The following two files can be created or edited to allow or restrict users from using the system's cron file:

- **/etc/cron.allow** – if **cron.allow** exists, it should contain a user's name to permit them to use cron jobs.
- **/etc/cron.deny** – if **cron.allow** doesn't exist but **cron.deny** does, the user who wants to use cron jobs must not be listed within the file.