YAML Script

🔷 Phase 1: Master the Basics of YAML

Topics to Learn:

- Syntax: indentation, comments
- Data types: strings, numbers, booleans
- Collections: lists, dictionaries/maps
- Nested structures
- Anchors and aliases (`&` and)
- Multi-line strings (`|` and `>`)

Practice: Create simple YAML files for:

- A to-do list
- A recipe
- A server configuration with nested objects

🔷 Phase 2: Apply YAML to DevOps Tools

Goal: Use YAML in real-world DevOps scenarios.

1. Docker Compose

- Learn how services, volumes, and networks are defined in `docker-compose.yml`
- **Practice**: Create a YAML file to spin up a multi-container app (e.g., Nginx + Redis + a web app)

2. GitHub Actions

- Understand `.github/workflows/*.yml` files
- **Practice**: Create a CI workflow that runs tests on push

3. Ansible

- Understand playbooks and inventory files
- **Practice**: Write a playbook to install NGINX and copy a config file

4. Kubernetes

- Learn to write YAML for:
  - Pods
  - Deployments
  - Services
  - ConfigMaps and Secrets
- **Practice**: Deploy a sample app using `kubectl apply -f deployment.yaml`

🔷 Phase 3: Real Projects & Challenges

Goal: Reinforce skills with real use cases.

Projects:

- Create a GitHub Action to build and deploy a Docker container to Docker Hub
- Write an Ansible playbook to configure a web server on AWS EC2
- Create Kubernetes manifests to deploy a Node.js or Python app

Challenges:

- Use YAML to define infrastructure as code with tools like **Terraform Cloud** (optional, advanced)
- Contribute to an open-source repo that uses YAML

🔷 Phase 4: Tools and Best Practices

- Use linters: YAML Lint
- Learn schema validation (e.g., K8s uses OpenAPI schema)
- Follow naming conventions and comments
- Automate checks in CI

Suggested Timeline (2–4 weeks)

Week	Focus
1	YAML basics + practice
2	Docker Compose + GitHub Actions
3	Ansible + Kubernetes
4	Projects + real-world examples

Examples for Each Section

🔷 1. YAML Basics

Task: Learn how YAML works

Example:

```
name: John Doe
age: 29
languages:
  - Python
  - Bash
  - YAML
contact:
  email: john@example.com
  phone: "123-456-7890"
```

Practice Idea: Write a YAML file to describe your DevOps toolset and skills.

🔷 2. Docker Compose Example

Task: Create a `docker-compose.yml`

Example:

```
version: '3.8'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
  redis:
    image: redis
```

Practice Idea: Add a volume and environment variable to the Redis service.

🔷 3. GitHub Actions Workflow

Task: Create `.github/workflows/ci.yml`

Example:

```
name: CI Pipeline
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
```

```
    - uses: actions/checkout@v3
    - name: Install Node
      uses: actions/setup-node@v3
      with:
        node-version: '18'
    - run: npm install && npm test
```

Practice Idea: Trigger the workflow only when pushing to `main`.

🔷 4. Ansible Playbook

Task: Write `webserver.yml`

Example:

```
- name: Install NGINX
  hosts: webservers
  become: yes
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: present
```

Practice Idea: Add a task to copy a custom Nginx config.

🔷 5. Kubernetes Deployment

Task: Write `deployment.yaml`

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app
          image: node:18
          ports:
            - containerPort: 3000
```

Practice Idea: Add a service and a ConfigMap.

Basic YAML Syntax

YAML (YAML Ain't Markup Language) is a human-readable data serialization language commonly used for configuration files, automation scripts, and data exchange between languages.

Key Concepts of YAML

- **Indentation**: Used to define structure (spaces only, no tabs).
- **Key-value pairs**: Represented as `key: value`.
- **Lists/Arrays**: Use hyphens to denote list items.
- **Comments**: Start with a `#`.

- **Data Types**: Supports strings, numbers, booleans, lists, and maps (dictionaries).

1️⃣ Scalar Values (Basic Data)

These are simple values like strings, numbers, or booleans.

```
name: John Doe
age: 35
is_student: false
```

> 💡 Note: Boolean values can be written as true/false, True/False, or yes/no.

2️⃣ Strings

Strings can be written without quotes. If needed, you can use single `' '` or double `" "` quotes.

```
message: Hello, world!
greeting: "Welcome to YAML!"
path: 'C:\\windows\\path'
```

Multi-line Strings

Use the pipe `|` to preserve newlines:

```
bio: |
  John is a software developer.
  He loves writing clean code.
  And automating things with YAML.
```

Use the greater-than `>` to fold newlines into a space:

```
bio: >
  John is a software developer.
  He loves writing clean code.
  And automating things with YAML.
```

This becomes:

`John is a software developer. He loves writing clean code. And automating things with YAML.`

3️⃣ Lists / Arrays

Use hyphens `-` to represent each item in a list.

```
hobbies:
  - Reading
  - Coding
  - Hiking
```

Nested lists are also possible:

```
students:
  - name: Alice
    grades:
      - 90
      - 85
      - 92
  - name: Bob
    grades:
      - 78
      - 88
      - 80
```

4️⃣ Maps / Dictionaries

Maps represent key-value pairs inside another key.

```
person:
  name: Jane Doe
  age: 28
  address:
    street: 123 Main St
    city: New York
    zip: 10001
```

5️⃣ Combining Lists and Maps

You can mix lists and maps together to create complex structures.

```
employees:
  - name: Sarah
    role: Developer
    skills:
      - Python
      - YAML
      - Docker
  - name: Mike
    role: Designer
    skills:
      - Figma
      - Sketch
```

6️⃣ Comments

Use the `#` symbol to add comments.

```
# This is a comment
name: John Smith  # Inline comment
```

7️⃣ Anchors and Aliases (Reusable Blocks)

Anchors (`&`) and aliases (`*`) allow you to reuse content.

```
default_config: &default
  environment: production
  timeout: 30s

service_a:
  <<: *default
  port: 8080

service_b:
  <<: *default
  port: 8000
```

This is equivalent to:

```
service_a:
  environment: production
  timeout: 30s
  port: 8080

service_b:
  environment: production
  timeout: 30s
  port: 8000
```

Common Pitfalls

Mistake	Explanation
Using tabs instead of spaces	YAML **requires** spaces for indentation. Tabs are not allowed.
Incorrect indentation	Indentation defines structure — even one space off can break your file.

Missing colon after key	Always use `key: value` format. A missing colon causes an error.
Misusing quotes	While optional, sometimes special characters need quotes to be interpreted correctly.

Valid YAML Example

Here's a complete example combining everything we've covered:

```
# Application Configuration File

app_name: My Awesome App
version: 1.0.0
debug_mode: true

database:
  host: localhost
  port: 5432
  credentials:
    username: admin
    password: securepassword123

features:
  - auth
  - payments
  - analytics

logging:
  level: debug
  output: |
    Logs will be written to
    this multi-line string.

deployments:
  - name: staging
    url: <https://staging.myapp.com>
  - name: production
    url: <https://myapp.com>
```

Tools to Validate YAML

- YAML Lint
- `yamllint` (CLI tool)
- IDE extensions (VS Code, PyCharm, etc.)
- `kubeval` (for Kubernetes YAML validation)

Summary

Feature	Syntax Example
Key-Value	`key: value`
String	`name: John`
Number	`age: 25`
Boolean	`active: true`
List	`- item1`
Map	`user: { name: Alice }`
Multi-line String	`text:
Comment	`# This is a com`

Understanding YAML Tags

What Are Tags?

In YAML, a **tag** is a prefix that tells the parser what **data type** or **schema** a value should conform to. Tags start with an exclamation mark `!` or use a URI-based namespace like `tag:yaml.org,2002:`.

They look like this:

```
value: !!str 123
```

Here, `!!str` means "interpret the value `123` as a string", even though it looks like a number.

Built-in YAML Tags (Core Schema)

YAML defines several built-in tags for common data types under the core schema:

Tag	Data Type	Description
`!!null`	Null	Represents a null value (`~` or `null`)
`!!bool`	Boolean	`true` , `false` , etc.
`!!int`	Integer	Whole numbers
`!!float`	Floating-point number	Decimal values
`!!binary`	Binary data	Base64-encoded strings
`!!timestamp`	Date/time	ISO 8601 format dates
`!!omap`	Ordered Map	A list of key-value pairs
`!!seq`	Sequence	List/array
`!!map`	Mapping	Dictionary/object
`!!str`	String	Text

Examples of Using Tags

1. Forcing a Number to Be a String

```
port: !!str 8080
```

Even though `8080` looks like a number, it will be interpreted as a string.

2. Specifying a Boolean Explicitly

```
active: !!bool "true"
```

This ensures `"true"` (as a string) is parsed as a boolean.

3. Using Timestamps

```
created_at: !!timestamp '2025-04-05T12:30:00Z'
```

Parsed as a date-time object if supported by the parser.

4. Binary Data

```
photo: !!binary |
  R0lGODdhDQAIAIAAAAAAANnxBvIAAAAC
```

Used to represent binary content like images in Base64.

Custom Tags

You can define your own tags to support **custom data types**, especially useful in domain-specific applications or APIs.

Example:

```
user: !User
  name: Alice
  role: admin
```

Here, `!User` is a custom tag that tells the parser to handle this block using a specific constructor or handler for `User` objects.

> 🔍 Note: Support for custom tags depends on the YAML parser and language. Not all libraries support them out of the box.

Use Cases for Tags

- Ensuring correct typing in configuration files.
- Supporting custom classes/objects in complex apps.
- Working with binary or structured data.

- Avoiding ambiguity in values like `"yes"`, `"on"`, or numeric strings.

Important Notes

- **Tags are optional**: Most YAML files don't use them because parsers infer types automatically.
- **Parser-dependent behavior**: Different YAML implementations may handle tags differently.
- **Avoid overuse**: Only use tags when necessary — they can make files harder to read.

Example Without vs With Tags

Without tags:

```
value: 123
```

With tag:

```
value: !!str 123
```

The first might be parsed as an integer; the second will always be a string.

Summary Table

Tag	Meaning	Example
`!!str`	String	`!!str 123` → `"123"`
`!!int`	Integer	`!!int "123"` → `123`
`!!float`	Float	`!!float "3.14"` → `3.14`
`!!bool`	Boolean	`!!bool "true"` → `true`
`!!null`	Null	`!!null ""` → `null`
`!!timestamp`	Date	`!!timestamp '2025-04-05'`
`!CustomType`	Custom	`!User { name: Alice }`

If you're working with tools like **Kubernetes**, **Ansible**, or **GitHub Actions**, you probably won't need tags often, but they're very useful in more advanced scenarios or when integrating with systems that expect strict data typing.

Let me know if you'd like examples using custom tags in Python (with PyYAML), or how to safely use tags in a real-world project!

💡 Adding - - - (3 dashes) unable us to write more than one YAML File in the same file

This is especially useful when you want to group related configurations, for example, defining multiple Kubernetes resources in one file, or organizing different configuration blocks without splitting them into separate files.

Example of Multi-document YAML Using `--`

Here's a simple example showing two separate YAML documents in a single file:

```
---
# First YAML document
name: Alice
role: Developer
skills:
  - Python
  - YAML
  - Docker

---
# Second YAML document
name: Bob
role: Designer
tools:
  - Figma
  - Sketch
```

In this example:

- The first document describes a developer.
- The second document describes a designer.
- Each is separated by `--`.

A proper YAML parser that supports multi-document mode will read these as two separate objects or maps.

Real-World Use Case: Kubernetes

One of the most common real-world uses of multi-document YAML is in **Kubernetes**, where you can define multiple resource types (like a `Deployment` and a `Service`) in a single file:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

This file defines both a **Deployment** and a **Service**, which can be applied together using `kubectl apply -f yourfile.yaml`.

Important Notes

- The `--` separator must start at the **beginning of a line**, not indented.
- After the first document, each new document starts with `--`.
- Some tools may require special support to parse multiple YAML documents from a single file.
- Not all parsers handle multi-document YAML by default — check documentation if you're using a specific tool or library.

Benefits of Using Multiple YAML Documents in One File

- **Organize related resources**: Especially helpful in Kubernetes, Ansible, etc.
- **Reduce file clutter**: Combine logically connected configurations.

- **Simplify deployment**: Apply or share a single file that contains multiple components.

How to Read Multi-document YAML in Code?

If you're working with code (e.g., Python), libraries like **PyYAML** or **ruamel.yaml** can load all documents from a single file.

Example in Python (using `ruamel.yaml`):

```
from ruamel.yaml import YAML

yaml = YAML()
with open("multi_doc.yaml") as f:
    docs = list(yaml.load_all(f))

for doc in docs:
    print(doc)
```

This would output both documents as separate dictionaries.

Summary

Feature	Description
`---`	Starts a new YAML document in the same file
Use Cases	Kubernetes manifests, grouped configs, logical grouping
Tools Support	Kubernetes (`kubectl`), Ansible, `ruamel.yaml` , etc.
Caution	Ensure correct formatting; some tools may not support multi-doc by default

YAML Data Structures Overview

Before diving into sequences and collections, let's quickly review the main types of data you can use in YAML:

Type	Description	Example
Scalar	A single value (string, number, boolean)	`name: John`
Sequence	An ordered list of values (like an array)	`- apple`, `- banana`
Mapping	A set of key-value pairs (like a dictionary or object)	`age: 30`
Collection	A general term that includes both sequences and mappings	

Sequence (List)

A **sequence** is an **ordered list** of items. In YAML, each item in a sequence starts with a hyphen and space (`-`).

Basic Sequence Example

```
fruits:
  - Apple
  - Banana
  - Orange
```

This represents a list of fruits.

Nested Sequences

You can have sequences inside other sequences:

```
shopping_list:
  - Fruits:
      - Apple
      - Banana
  - Drinks:
      - Water
      - Juice
```

Mapping (Dictionary/Object)

A **mapping** is a collection of **key-value pairs**, also known as a hash, dictionary, or object in other languages.

Basic Mapping Example

```
person:
  name: Alice
  age: 25
  is_student: true
```

Each line under `person`: is a key-value pair.

Nested Mappings

You can nest mappings inside other mappings:

```
employee:
  name: Bob
  job:
    title: Developer
    department: Engineering
```

Collection: The General Term

The term **collection** in YAML refers to any structured group of data, whether it's a **sequence** (list) or a **mapping** (dictionary). It's not a specific syntax but a **conceptual grouping**.

So:

- If you're working with a list of things → you're using a **sequence** (a type of collection).
- If you're working with key-value pairs → you're using a **mapping** (also a type of collection).

Mixed Collections Example

```
users:
  - name: Alice
    role: Admin
  - name: Bob
    role: Editor
```

Here:

- `users` is a **sequence** (a list of user objects).
- Each user is a **mapping** (has keys like `name` and `role`).
- So overall, this is a **collection of collections**!

Inline Format (Block Style)

YAML supports **inline formatting** for sequences and mappings using square brackets `[]` and curly braces `{}`, respectively.

Inline Sequence

```
fruits: [Apple, Banana, Orange]
```

Inline Mapping

```
person: {name: Alice, age: 25, is_student: true}
```

> ⚠️ Tip: While inline format saves space, it can reduce readability. Use it for short or simple structures only.

Combining Sequences and Mappings

In real-world YAML files (especially in tools like Kubernetes, Ansible, GitHub Actions), you'll often see complex combinations of sequences and mappings.

Example: Complex Collection

```yaml
servers:
  - name: Web Server
    ip: 192.168.1.10
    services:
      - nginx
      - ssl
  - name: DB Server
    ip: 192.168.1.11
    services:
      - mysql
      - backup
```

Breakdown:

- `servers`: A **sequence** of server definitions.
- Each server is a **mapping** with keys `name`, `ip`, and `services`.
- `services`: A **sequence** of service names.

Summary Table

Structure	Syntax	Meaning	Example
Scalar	`key: value`	Single value	`name: John`
Sequence	`- item`	Ordered list	`- Apple`, `- Banana`
Mapping	`key: value`	Key-value pairs	`age: 30`
Collection	N/A	General term for sequences and mappings	Used together in complex structures

---

Best Practices

- Use indentation consistently (spaces only, no tabs).
- Prefer block style (`item`) over inline (`[item1, item2]`) for better readability.

- Nest mappings and sequences logically.
- Combine sequences and mappings to model real-world data.

In YAML, you can **mix block and flow styles** in the same document to make your configuration **readable** and **concise**. This is especially useful when dealing with complex data structures like lists of maps or nested key-value pairs.

Let's look at **two real-world examples** where **block style** (using indentation and hyphens) and **flow style** (inline syntax using `{}` for mappings and `[]` for sequences) are used together effectively.

Example 1: User Roles Configuration

This example shows a list of users (`block style sequence`) where each user has a name and a list of roles (`flow style sequence`).

```
users:
  - name: Alice
    roles: [admin, editor]
  - name: Bob
    roles: [editor]
  - name: Charlie
    roles: [viewer]
```

Breakdown:

- The overall structure uses a **block style** for the list of users.
- Each user is a mapping with keys `name` and `roles`.
- The `roles` field uses a **flow style** for compactness and clarity.

This format keeps the file clean while clearly expressing structured data.

Example 2: Kubernetes Deployment with Environment Variables

This example shows a Kubernetes-style deployment where environment variables are defined using a **flow style** inside a **block-style** resource definition.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: app-container
          image: my-app:latest
          env:
            - name: ENV1
              value: "value1"
            - name: ENV2
              value: "value2"
        - name: sidecar-container
          image: sidecar:latest
          env: [{ name: DEBUG, value: "true" }, { name: LOG_LEVEL, value: "debu
g" }]
```

Breakdown:

- The entire deployment is written in **block style** for readability.
- The second container's `env` field uses **flow style** for brevity.
- You get the best of both worlds: clarity for the main structure, compactness for repeated simple entries.

When to Use Block vs Flow Style

Style	When to Use	Pros	Cons
Block	Complex structures, deep nesting, better readability	Easy to read and edit	More verbose
Flow	Simple lists/mappings, short inline values	Compact, saves space	Can become hard to read if too complex

Sequence of Mapping

```
users
 - name: John Doe
   age: 30
   email: john.doe@example.com

 - name: Jane Smith
   age: 25
   email: jane.smith@example.com
```

Mapping of Sequences

Block Style

```
departments:
 engineers:
  - Alice
  - Bob
  - Carol

marketing:
 - Dave
 - Eve
 - Frank
```

Flow Style

```
departments: {engineers: [Alice, Bob, Carol], marketing: [Dave,Eve,Frank]}
```

Sets

Store a unique collection of values

```
cities: !!set
 ? Washington DC
 ? London
 ? Delhi
```

Ordered Mapping

Maintain a list of elements and their ordering

```
capitals: !!omap
 - USA: Washington DC
 - United Kingdon: London
 - India: Delhi
```

Complex Key

Always start with a ?

```
? Which is your favourite sport when you were a child ?
: Cricket
? |
Which is your favourite sport when you were a child ?
: Cricket
? - Newyork
    - Chicago
    - Washington
    : USA
    #a list that its values is a list
? - Development
  - UAT
  - Production
 : - com
   - edu
   - jo
```

💡 Each key in YAML should be unique

Anchors and Aliases in YAML

Anchors and *aliases* are powerful features that help avoid repetition and make configuration files cleaner and more maintainable.

What Are Anchors and Aliases?

- **Anchors (`&`)** allow you to define a section of content that you want to reuse.
- **Aliases (*)** let you reference that anchored content elsewhere in the YAML file.

This is especially useful when you have repeated configurations, such as similar containers, environment variables, or settings.

Syntax and Example

```
defaults: &default_settings
  retries: 3
  timeout: 30
  log_level: info

service_a:
  <<: *default_settings
  timeout: 60

service_b:
  <<: *default_settings
```

Breakdown:

- `&default_settings` defines an anchor named `default_settings` under the `defaults` key.
- `default_settings` is an alias that reuses the content of the anchor.
- `<<: *default_settings` is YAML's **merge key**, which merges the referenced values into the current level.
- `service_a` overrides the `timeout` from the default.

Without Merge Key

You can also use aliases without the merge key, which helps reuse scalar values, lists, or dictionaries.

```
primary_color: &main_color "#FF5733"

header:
  background: *main_color
```

```
footer:
  background: *main_color
```

Benefits

- **DRY principle** (Don't Repeat Yourself): Reduces duplication.
- Easier to update: Change one anchor and it reflects everywhere it's used.
- Cleaner configuration for complex environments.

Cautions

- Be careful not to overuse aliases, especially in very large files — they can make the configuration harder to understand.
- Not all parsers or tools support YAML 1.1 or 1.2 features equally, so test carefully.

💡 It is not supported in JSON

Overriding Anchors and Aliases in YAML

One of the powerful features of using **anchors (`&`)** and **aliases (`*`)** in YAML is that you can **override specific values** when reusing a configuration.

How It Works

When you use the **merge key (`<<`)** to import values from an alias, you can **override individual keys** in the same block by redefining them *after* the merge.

Example: Overriding Values

```
defaults: &default_config
  retries: 3
  timeout: 30
  log_level: info

service_a:
  <<: *default_config
  timeout: 60  # overrides the value from defaults

service_b:
  <<: *default_config
  log_level: debug  # overrides log_level
```

Explanation

- `service_a` and `service_b` both start by pulling in values from `default_config` using the merge key (`<<: *default_config`).
- Then they **override** specific values:
  - `service_a` overrides `timeout`.
  - `service_b` overrides `log_level`.

YAML processes values **top-down**, so the override happens because the new value comes *after* the merge.

What You Can Override

You can override:

- **Scalars** (e.g., strings, numbers, booleans)
- **Keys in dictionaries/maps**

You **cannot** override:

- Anchors directly (you can only override where they are used)
- Lists in a way that merges them — you'd have to redefine the list entirely

Overriding Lists (Manually)

YAML does **not** support merging lists via aliases. If you want to change a list, you must write it out completely:

```
default_list: &common_list
  - apple
  - banana

custom_list:
  items:
    - orange  # completely replaces the list, not merged
```

Two practical DevOps use cases showing anchors, aliases, and overriding values in YAML:

1. Docker Compose Example

```
version: "3.9"

x-common-settings: &default_service
  restart: always
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"
```

```
services:
  app:
    <<: *default_service
    image: my-app:latest
    ports:
      - "8080:80"

  worker:
    <<: *default_service
    image: my-worker:latest
    environment:
      - QUEUE=default
    logging:
      driver: "json-file"
      options:
        max-size: "50m"  # 🔁 overridden only for worker
        max-file: "5"    # 🔁 overridden only for worker
```

Key Points:

- Both `app` and `worker` services inherit the common settings via `<<: *default_service`.
- The `worker` service **overrides** logging options (`max-size`, `max-file`) to suit its heavier workload.

---

2. Kubernetes Deployment Example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 2
  template:
    metadata:
```

```
    labels: &app_labels
      app: myapp
      tier: frontend
  spec:
    containers:
      - name: web
        image: nginx:latest
        ports:
          - containerPort: 80
        env:
          - name: ENVIRONMENT
            value: production

---
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector: *app_labels  # 🔁 reuses the same labels as in deployment
  ports:
    - port: 80
      targetPort: 80
```

Key Points:

- The labels in the Deployment pod spec are given an **anchor** `&app_labels`.
- The Service's selector uses the **alias** `app_labels` to match the same labels.
- This ensures both resources are always in sync.

Bonus: Combine Anchors and Overrides

In both examples, you can override just the fields you need while inheriting most values from a common definition, saving time and reducing errors.

Multiple Documents in YAML

YAML allows you to define **multiple documents** in a single file using the `---` separator. This is useful when you want to bundle related configurations together, like multiple Kubernetes resources or Docker Compose overrides.

Syntax

```
 # First document
---
name: document_one
type: config
enabled: true

# Second document
---
name: document_two
type: override
enabled: false
```

- Each document **starts** with `--` (optional for the first document but required for the others).
- YAML parsers will treat them as **separate documents**.

Example: Kubernetes with Multiple Resources

```
# Deployment
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myapp:latest
          ports:
            - containerPort: 80

# Service
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - port: 80
      targetPort: 80
```

Use Cases

Use Case	Example
Kubernetes configs	Deployments, Services, PVCs
Docker Compose extensions	Base + overrides
CI/CD workflows	Multiple jobs or stages

CloudInit bootstrapping	Script + cloud config blocks

Tips

- YAML parsers like `kubectl`, `yq`, and CI tools like GitHub Actions support multi-document files.
- You can **split or extract** documents using tools like `yq`, `sed`, or custom scripts.