DEPLOYMENT_ESSENTIALS

> The cloud is just someone else's computer.

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

- How to Deploy Your Website
  - **React** ;for pure frontend app; no SSR
    - **Vercel**
    - **Virtual machines** ;Digital Ocean, AWS, GCP, Azure
    - **CDNs + object store** ;most optimal; for pure frontend app; no SSR
  - **NextJS / Node.js / Golang / Rust / JAVA**
    - **Cheaper serverless**
    - **Virtual machines**
    - **Self hosted in your home lab**
    - **Autoscaled servers**
  - **Dockerised app** OPTIMAL
    - **Kubernetes cluster**
    - **Decentralised compute 1st** (This is a less common term, suggesting a focus on decentralized infrastructure.) Immature; but cheaper
    - **Open Source PaaS/vercel - coolify**

Deployment Decision Matrix

For Static React Applications:

Method	Cost	Scalability	Complexity	Best For
Vercel	High	Excellent	Low	Rapid prototyping
VM + Nginx	Medium	Limited	Medium	Learning/small projects
CDN + Object Store	Low	Excellent	Low	**Production recommended**

For Full-Stack Applications:

Method	Cost	Scalability	Complexity	Best For
Vercel/Netlify	High	Excellent	Low	Rapid development
Serverless	Variable	Excellent	Medium	Variable traffic
VM + Auto-scaling	Medium	Good	High	Predictable traffic
Kubernetes	Variable	Excellent	High	**Enterprise production**

Table of Contents

---

1. Introduction and Context

This document discusses **10+ different ways to deploy applications at Real-world Production Level**. The applications considered include:

- A simple **React website** (pure front-end).
- A slightly more complicated front-end using **Next.js**.
- A backend in **Node.js, Golang, Rust, or Java**.
- A **containerized application** (which could be anything containerized).

Deployment methods range from:

- **Beginner friendly ways** (e.g., Vercel).
- **Scalable and price effective ways** (e.g., serverless or hosting in a Kubernetes cluster).
- **Completely dystopian ways** (e.g., hosting on a local machine exposed to the internet or decentralized compute).

The goal is to discuss the pros and cons of each method and determine the right way to deploy an application based on the company's scale.

2. Part 1: Deploying a Pure Front-End Application (React)

The first part involves creating a simple **React application** (a pure front-end application).

2.1. React Application Setup

The initial application setup involves several steps:

1. Initialize a React application.
2. Open it in Visual Studio Code.
3. Install all the dependencies.
4. Run it locally to verify functionality.

A simple front-end only application structure was created with four components:

1. A **nav bar** (at the top).
2. A **hero section** (rendering a video).

3. A **new courses section** (showing a list of upcoming courses).
4. **Existing courses** (showing existing courses).

Styling: Tailwind CSS was added to make the application prettier and ensure it looks better. https://v2.tailwindcss.com/docs/installation

2.2. How React Works and Compiling for Deployment

React is a framework that ultimately outputs **HTML, CSS, and JavaScript files** which must be deployed on the internet.

- JSX files, TSX files, and components (e.g., `button.TSX`) are for the developer to write reusable code and are **not understood by the browser**.
- Browsers only understand HTML, CSS, and JavaScript files.

Transpilation/Conversion: The React code must be converted to HTML, CSS, and JavaScript.

- Most frameworks used to bootstrap a React application (e.g., **Vite**) provide a command to transpile TSX files into JavaScript.
- The command used is `npm run build`.

Distribution Folder (`dist`):

```
npm run build
```

This command creates a `dist` folder containing:

- `index.html` - Main HTML file
- CSS files - Styling
- JavaScript files - Application logic

> The original React code (JSX/TSX files) is only for developer convenience. Browsers only understand HTML, CSS, and JavaScript.

- Running `npm run build` generates a new folder called `dist` (distribution), which we distribute to in the world.
- The command logs confirm the creation of `index.html`, a CSS file, and a JavaScript file.
- The website can be deployed on the internet **as long as the `dist` folder exists**; the original source code is not needed.
- The `dist` folder contains the files that must be distributed and hosted on the internet.
- Some frameworks hide this complexity and convert the React code automatically, while sometimes it must be done manually.

2.3. Deployment Method 1: Platforms as a Service (PaaS) - Vercel

Vercel is the most popular Platform as a Service (PaaS), with **Netlify** being another option that offers similar functionality.

Vercel Workflow:

1. Go to `vercel.com` and sign up.
2. Push code to **GitHub**.
    - Sign-In in Github, Create New-Repo
    - **GitHub Steps Recap:**

    -
      ```
      git init
      git add .
      git commit -m "Initial commit"
      git remote add origin <your-github-repo-url> #
      git@github.com:username/project.git
      git push origin main #or HEAD
      ```
        - Initialize a Git repository (`git init`).
        - Add all files (`git add .`).
        - Commit changes (`git commit -m "init"`).
        - Push code to GitHub remote (`git remote add origin [URL]`, `git push origin main/head`).
3. In the Vercel dashboard, click **Add New > Project**.
4. Import the Git repository (requires linking GitHub to Vercel).
5. Vercel automatically detects the application type (e.g., React application created with Vite).
6. Set-up extra stuffs like domain, dir, env vars, etc.
7. Click **Deploy**.

Result: The website is deployed (e.g., `landing-page-react-steel.vercel.app`).

Domain Name Attachment

1. Buy and manage a domain name from a registrar (e.g., `godaddy.com`, `squarespace.com`, `Google domains.com`).
2. Go to the domain provider's DNS management section.
3. Go to Specific Domain's Manage section (e.g. `kintsugidev.studio`)
4. Go to Manage Custom Records
5. Add a custom record where the application should be hosted (e.g., `mynewproject`).
6. now new custom domain generated, for our eg: `mynewproject` gets new sub-custom-domain named `mynewproject.kintsugidev.studio`
7. Vercel requires a **CNAME** record pointing to `cname.vercel-dns.com`.
8. After waiting, the application is automatically deployed on the custom domain.

> CNAME (Canonical Name) record is a type of resource record in the Domain Name System (DNS). It's used to create an alias (another name) from one domain or subdomain name to another domain name.

> A CNAME record never points directly to an IP address. It must always point to another domain name. It delegates the responsibility of finding the IP address to the target name.

SSL Certificate

- Vercel automatically generates an **SSL certificate**.
- An SSL certificate provides HTTPS (secure) rather than HTTP (insecure).
- Free SSL Certificate Provider Links
  - https://letsencrypt.org/
  - https://www.sslforfree.com/
  - https://zerossl.com/

Automatic GitHub Integration (CI/CD)

- Vercel provides **automatic GitHub integration**.
- It offers **Continuous Integration/Continuous Deployment (CI/CD)** by continuously monitoring the GitHub repository.
- Whenever a commit occurs, Vercel automatically deploys the code.
- *Example:* If a component is removed and committed, Vercel initiates a build. If the build fails (e.g., due to an unremoved import statement), correcting the error and committing again triggers a successful rebuild and deployment.

PaaS Benefits:

- Zero configuration
- Automatic GitHub integration (Continuous Deployment)
- SSL certificates included
- Global CDN

PaaS Drawbacks:

- **Vercel is very expensive** because it provides ease of use and benefits like automatic deployment and SSL generation.
- Users might move away if the pricing structure is unsuitable.
- Limited control over infrastructure

2.4. Deployment Method 2: Virtual Machines (VMs)

Deploy React app on cloud virtual machines (AWS EC2, Digital Ocean, GCP).

> front-ends should ideally *not* be deployed this way, but the knowledge is useful, especially for backends.

- Why Virtual Machines?
  - Full control over environment
  - Cost-effective for consistent traffic
  - Custom server configurations

Cloud Providers and VMs

- Major cloud providers: **AWS** (Amazon Web Services), **GCP** (Google Cloud Platform), **Vultr**, **DigitalOcean**.
- Their primary offering is the **Virtual Machine** (VM).

- A VM is a virtual layer created on top of a very big pool of compute (RAM, CPU, memory) located in data centers globally.
- The VM grants access to allocated specifications (e.g., 2 CPUs, 4 GB RAM).

Public IP Requirement

- The VM provides a **public IP**.
- A public IP is necessary to run the application on the internet; local Mac machines generally cannot expose processes on the internet without complex methods like tunneling (discussed later) because public IPs are limited resources rented by cloud providers.

Deployment Steps on AWS EC2

AWS (using EC2, Elastic Compute 2) is used as an example, but steps are similar across cloud providers:

Step	Action	Details
1	**Get a Virtual Machine (VM)**	Select EC2 in AWS dashboard, click **Launch Instances**. Name the VM (e.g., `react app`), select OS (e.g., **Ubuntu**), select instance type (e.g., small: 1 vCPU, 2 GB memory). Network settings involve **setting up a firewall rule**.
	Firewall Configuration	Allow **HTTP** and **HTTPS** traffic. HTTP traffic uses **Port 80**, HTTPS uses **Port 443**. Port 5173 (Dev mode) can also be allowed. Set storage (e.g., 8GB).
2	**Install Node.js**	Connect to the running instance (e.g., using EC2 Instance Connect browser method or ideally key pairs). Install Node.js, typically using **Node Version Manager (NVM)** on Ubuntu. Commands involve running NVM installation script and then `NVM install node`.
3	**Copy Source Code**	Clone the repository using `git clone [repo URL]`. Navigate to the repo directory and run `npm install`.
4	**Start Application in Production Mode**	**Do not run `npm run Dev`** (this is Dev mode, uncompressed, exposes dev data). **Run `npm run build`** to convert source code to HTML, CSS, and JavaScript, generating the `dist` folder.

Serving Static Files

The `dist` folder files (`index.html`, CSS, JS) must now be served.

Serving Methods:

1. Use **Engine X** (a common HTTP server/reverse proxy) to serve files statically.
2. Use a library like **`serve`**.

Using `serve` library:

1. Globally install `serve`: `npm install -g serve`.
2. Run `serve` inside the `dist` folder. It defaults to hosting files on **Port 3000**.

3. **Firewall Update:** Since only ports 80 and 443 were opened initially, Port 3000 must be opened in the Security Group/inbound rules on the VM.
4. Once Port 3000 is open, the website can be accessed via `[Machine's IP]:3000`.

Connecting to a Domain (A Record)

1. Obtain the machine's public IP address.
2. Go to the domain provider's DNS management.
3. Add an entry (e.g., `react-d2.kintsugidev.com`) using an **A record** that points directly to the machine's IP address.
4. The site is now hosted on `react-d2.kintsugidev.com:3000`.

Serving on Default HTTP Port (Port 80)

The default HTTP port is **Port 80**. To remove the ugly `:3000` suffix, the application must be started on Port 80.

- Port 80 is a **provisioned port**, meaning not everyone has access to start processes on it.
- Running `serve` on Port 80 (`serve -p 80`) results in `error permission denied`.
- **Solution:** Install and run the application/serve library as the **super user** (`sudo`).

Super User Steps:

1. Update registry: `sudo apt update`.
2. Install serve globally as super user: `sudo npm install -g serve`.
3. Run serve on Port 80 as super user: `sudo serve -p 80`.

Result: The application is now accessible via the domain name without the port number (e.g., `react-2.kintsugidev.com`).

Steps:

1. Create EC2 Instance

- Choose Ubuntu OS
- Select t2.micro or t2.small
- Configure security groups (HTTP, HTTPS, SSH)

2. Install Node.js

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh |
bash
source ~/.bashrc
nvm install node
```

3. Deploy Application

```
git clone <your-repo-url>
cd <project-folder>
npm install
npm run build
```

4. Serve Static Files Using serve library:

```
sudo npm install -g serve
sudo serve -s dist -p 80
```

Using nginx (recommended):

```
sudo apt update
sudo apt install nginx
sudo cp dist/* /var/www/html/
sudo systemctl start nginx
```

5. Custom Domain

- Add A record: `subdomain` → `<server-ip>`
- Install SSL with Certbot:

```
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d yourdomain.com
```

SSL Certificate Assignment

The site is currently "not secure".

- **Ways to add SSL:** Buy a certificate or get one for free.
- The easiest free way is using **Certbot**.
- **Engine X as a reverse proxy** and **Certbot** to install an SSL certificate
- An alternative combination is **Apache and Certbot**. To make your site secure, add an SSL certificate using Certbot with either Nginx (Engine X) or Apache as your web server. The Certbot tool automates the process and provides a free certificate from Let’s Encrypt. Here’s a clear, step-by-step solution for both setups.

Nginx (Engine X) + Certbot

1. Install Nginx

```
sudo apt update
sudo apt install nginx
```

2. Allow HTTP and HTTPS in Firewall

```
sudo ufw allow 'Nginx Full'
```

3. Install Certbot and Nginx Plugin

```
sudo apt install certbot python3-certbot-nginx
```

4. Obtain and Install SSL Certificate

```
sudo certbot --nginx -d yourdomain.com -d www.yourdomain.com
```

- Certbot will automatically configure Nginx for SSL.
- Follow prompts and confirm domain ownership.

5. Test Automatic Renewal

```
sudo certbot renew --dry-run
```

6. Verify HTTPS

- Visit `https://yourdomain.com` in browser; the site should be secure.

Apache + Certbot

1. Install Apache

```
sudo apt update
sudo apt install apache2
```

2. Allow HTTP and HTTPS in Firewall

```
sudo ufw allow 'Apache Full'
```

3. Install Certbot and Apache Plugin

```
sudo apt install certbot python3-certbot-apache
```

4. Obtain and Install SSL Certificate

```
sudo certbot --apache -d yourdomain.com -d www.yourdomain.com
```

- Certbot will automatically configure Apache for SSL.

5. Test Automatic Renewal

```
sudo certbot renew --dry-run
```

6. Verify HTTPS

- Visit `https://yourdomain.com` in browser; the site should be secure.

Option	Server Setup	Certbot Plugin	Main Command	Notes
Nginx + Certbot	`nginx`	`python3-certbot-nginx`	`sudo certbot --nginx -d ...`	Easiest for most cases
Apache + Certbot	`apache2`	`python3-certbot-apache`	`sudo certbot --apache -d ...`	Alternative setup

Using Certbot automates the SSL process and ensures free, renewable certificates for your domain. Both server types are supported and documented for quick setup.

Benefits:

- Full control
- Predictable costs
- Custom configurations

Drawbacks:

- Manual setup required
- Single point of failure
- No automatic scaling

> Scaling Drawbacks of VMs: This method is **not good for scaling purposes** because it relies on a **single machine (single EC2 machine)**. A single machine cannot handle the load of a million users.

2.5. Deployment Method 3: Optimal Solution - CDN and Object Stores

This is the most ideal way to deploy a React application for **cost purposes and scaling purposes**.

- **Object Store**: Stores files (S3, Bunny Storage)
- **CDN**: Caches and delivers files globally (CloudFront, Bunny CDN)

Definitions

- **Object Stores:** Storage meant for holding files (big objects) like `index.html`, CSS, JavaScript, MP4 videos, and JPEG images.
  - **Popular Object Stores: S3** (AWS), **R2** (Cloudflare).
  - **Reason to use:** EC2 machines have network, file system, and run processes, while an object store's sole purpose is storage.
- **CDN (Content Delivery Network):** A network of thousands of machines (called **POPs** or Points of Presence) worldwide used for content delivery.
  - **Function:** When a user requests content (e.g., `index.html` or MP4), they hit their closest POP. The POP grabs the content from the source (e.g., Object Store or EC2 machine) and **caches** it. Subsequent users in that region receive the content directly from the cached POP.
  - **Benefit:** The file gets cached close to the user, reducing latency and load on the source server.
  - **Popular CDNs: CloudFront** (AWS), **Cloudflare CDN**, GCP CDN.

Why This Combination?

- **Infinite scalability** - CDN handles traffic spikes
- **Global performance** - Files served from nearest location
- **Cost-effective** - Pay only for storage and bandwidth
- **No server maintenance** required

Comparison of Approaches

Approach	Visibility	Downside	Scaling
Vercel (PaaS)	Zero visibility	Expensive, pricing can be problematic at high scale	Handles scaling automatically
Own Machine (VM/EC2)	Full ownership	Does not scale infinitely, requires a lot of compute	Poor scaling beyond starting multiple machines manually
CDN + Object Store	High control	N/A (Optimal for static content)	Scales almost infinitely due to caching

When to Use CDN + Object Store

This combination is ideal if:

1. **Serving videos** (e.g., YouTube).
2. **Serving images** (e.g., Instagram).
3. The website is **static** (it does not use server-side rendering).
    - **Static Website Definition:** Every user gets the same set of files from the server.

When NOT to Use CDN + Object Store

- When using **Next.js with Server Side Rendering (SSR)**.
- If every user gets a different `index.html` content, caching on the CDN is impossible.

Using Bunny.net (Object Store + CDN Example)

Bunny is a newer startup often slightly cheaper than AWS. It is used here because it is easier to deploy on. Price optimization is important if CDN pricing affects the company (e.g., serving a lot of images/videos), but usually not for simple HTML files.

Bunny Deployment Steps:

Step	Action
1	Sign up on Bunny.
2	**Create an Object Store (Storage Zone)**
3	**Upload Files**
4	**Create a CDN (Pull Zone)**
5	**Connect Domain to CDN**
6	**Activate SSL**

1. Create Object Store

- Sign up at bunny.net
- Create new Storage Zone
- Upload all files from `dist` folder

2. Create CDN

- Create new Pull Zone
- Connect to Storage Zone as origin
- Configure regions based on user location

3. Custom Domain

- Add CNAME record: `subdomain` → `<cdn-url>`
- Enable SSL (automatic)

Result: The application is hosted on `https://react-d3.kintsugidev.studio` via a CDN.

Scaling: This method scales well, often handling millions of users easily due to caching, especially when serving static content like HTML and JavaScript files.

3. Part 2: Deploying Backend Applications (Next.js Example)

Backend applications are fundamentally different from static front-ends and **cannot be scaled via a CDN**.

Next.js applications include both frontend and backend functionality, requiring different deployment strategies.

Unlike static React apps, backend applications:

- Cannot use CDN caching (each request is unique)
- Require server processing
- Need database connections
- Must handle dynamic content

3.1. Backend Scaling Limitation with CDN

- When serving backend data (like JSON or XML), the content is typically user-specific (e.g., an Instagram feed).
- If a CDN caches the response for User 1, User 2 would mistakenly receive User 1's cached data.
- **Every backend request must go directly to the source** (e.g., an EC2 server).
- While edge servers (multiple EC2 servers in various regions) can be used, data cannot be cached using a CDN for user-specific data.

3.2. Next.js Application Setup (Front-end + Backend)

The React application is converted to a Next.js application.

- Next.js allows for **server-side rendering** (SSR) and backend routes.

Backend Endpoint Example:

- A pre-register submit button is configured to hit a backend endpoint: `/api/pre-register`.
- This route is created within the Next.js project structure (`/app/api/pre-register/route.ts`).
- The `post` function in this file gets the user's body (email), logs it to the screen, and returns "done". (Ideally, this data should be stored in a database).

3.3. Deployment Method 4: Cheaper Serverless Alternatives (Cloudflare Pages)

While Next.js applications can be deployed easily on **Vercel** (who owns Next.js), Vercel becomes **very expensive** quickly when actively used.

Cheaper Alternatives:

- **Netlify**.
- **Cloudflare Pages**.

Pricing Comparison (Real-World Experiments):

- Vercel: $720 bill in a week.

- Netlify: $300 bill in a week.
- Cloudflare: Worked free for a really long time.

Cloudflare Offerings:

- **Pages:** For applications with both front-end and backend functionality.
- **Workers:** Serverless offering for pure backend applications, noted for being fast and cheap.
- **Integration:** Cloudflare Pages does not offer direct Git integration out of the box; GitHub pipelines might need to be written manually.

Steps:

```
npm create cloudflare@latest my-app
cd my-app
npm run build
npm run deploy
```

Configuration for Next.js:

Add to API routes:

```
export const runtime = 'edge';
```

Cloudflare Deployment Steps:

1. **Sign Up:** Go to `cloudflare.com` and sign up.
2. **Initialize Project:** Use the CLI command:

   ```
   npm create cloudflare@latest
   ```

   (Select defaults, and choose Next.js as the framework).
3. **Deploy Basic App:** Deploy the basic app via the CLI if prompted.
4. **Replace Source Code:** Update the files (app folder, components, public assets/images) in the Cloudflare adapter Next.js project with the custom landing page code.
5. **Build (Local):** Run `npm run build` locally to check for compilation errors (Next.js can be stricter than React).
6. **Edge Runtime Configuration:** Deploying often requires exporting `Edge runtime route segment config` for non-static routes (like the backend API route `/api/pre-register`).
7. **Deploy:** Run `npm run deploy` (which executes `npm run pages:build` and `wrangler pages deploy`).

Result: The application is deployed on Cloudflare Pages (e.g., `next-kintsugidev-app.pages.dev`).

Domain and SSL: A custom domain (e.g., `next-1.kinstugidev.studio`) can be set up in the Cloudflare dashboard, requiring a CNAME entry in the DNS provider. Cloudflare automatically provides an SSL certificate.

Benefits:

- Automatic scaling
- No server management
- Pay-per-request pricing
- Global edge deployment

Drawbacks:

- Vendor lock-in
- Cold start delays
- Limited runtime environments

3.4. Deployment Method 5: Virtual Machines (VMs) - Next.js

Deploying a Next.js application on a VM is similar to deploying React.

Steps Recap (AWS EC2 used):

1. **Create VM:** Launch an EC2 instance (e.g., `next app`, Ubuntu OS, medium size, 8GB storage). Allow HTTP traffic.
2. **Prepare Code:** Ensure the Next.js app builds locally and push code to a public GitHub repository.
3. **Connect and Install:** Connect to the instance via EC2 Instance Connect. Install Node.js using NVM.
4. **Clone and Install Dependencies:** Clone the repo and run `npm install`.
5. **Run in Production:**
    - Run `npm run build` (creates the `.next` folder).
    - Run `npm run start` (starts the application on `Localhost:3000` in production mode).
6. **Open Port:** Update the Security Group inbound rules to allow custom TCP traffic on **Port 3000** from anywhere.
7. **Access:** The application is available at `[Machine's IP]:3000`.
8. **Connect Domain:** Add an **A record** in the DNS provider pointing the desired domain (e.g., `next-2.kintsugidev.com`) to the server's IP.

Improving Deployment

The site is accessed via `domain:3000` and is "not secure".

- **Reverse Proxy/Port 80/443:** To remove the `:3000`, the app should run on Port 80 (HTTP) or Port 443 (HTTPS).
    - **Recommendation:** Use a reverse proxy (like **Engine X**) running on Port 443, which forwards requests to the application process running on Port 3000.
    - Starting the Next.js app directly on Port 80 or 443 is generally not recommended.
- **SSL Certificate:** Needed for security.
    - Use **Certbot** for free SSL certificates, or buy one.

    - This is left as an assignment: Figure out how to use a reverse proxy plus Certbot, or how to start the app directly on Port 443/80.

To eliminate the `:3000` from the URL and secure your site, use Nginx as a reverse proxy on ports 80/443, forwarding traffic to the app running on port 3000, and install an SSL certificate with Certbot. Here's a clear, step-by-step solution .

Solution: Removing Port and Adding SSL

1. Next.js App on Port 3000

Start your app as usual on port 3000:

```
npm run start
##### App listens on http://localhost:3000
```

2. Install Nginx as Reverse Proxy

```
sudo apt update
sudo apt install nginx
```

3. Configure Nginx Reverse Proxy

Edit (or create) your site config:

```
sudo nano /etc/nginx/sites-available/yourdomain.com
```

Paste:

```
server {
    listen 80;
    server_name yourdomain.com www.yourdomain.com;

    location / {
        proxy_pass          http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header    Upgrade $http_upgrade;
        proxy_set_header    Connection 'upgrade';
        proxy_set_header    Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Enable the config:

```
sudo ln -s /etc/nginx/sites-available/yourdomain.com /etc/nginx/sites-
enabled/
sudo nginx -t
sudo systemctl reload nginx
```

Now, visiting `http://yourdomain.com` (without `:3000`) will forward requests to your app.

4. Install SSL Certificate with Certbot

Install Certbot and Nginx plugin:

```
sudo apt install certbot python3-certbot-nginx
```

Run Certbot:

```
sudo certbot --nginx -d yourdomain.com -d www.yourdomain.com
```

This adds HTTPS support and auto-configures Nginx.

5. Confirm HTTPS

- Visit `https://yourdomain.com` (no `:3000`) and verify the site is secure.

Alternative (Not Recommended)

You may run your app directly on port 80/443 using root privileges or setcap, but **using a reverse proxy (Nginx) is best practice** for security, flexibility, and SSL management.

Summary Table :Improving Deployment

What	How
Remove `:3000`	Nginx reverse proxy on 80/443 to 3000
Add SSL/HTTPS	Certbot with Nginx for free SSL cert
Secure Deployment	Access via `https://yourdomain.com`

This method is recommended for production and works for all modern Node.js (including Next.js) web applications.

3.5. Deployment Method 6: Self-Hosting with Tunnels in a Home Lab (Dystopian Scenario)

This is something that **should probably not be done** but is discussed for adventurous use cases (e.g., exposing CCTV footage from a home lab).

Goal: Route traffic from a public URL (e.g., `next-3.kintsugidev.studio`) directly to a local machine (Mac machine) where the Next.js server is running.

Downsides:

- The local machine is **unreliable**.
- Exposing the machine to the world is a security risk.
- Internet or power outages can take the application down.
- Applications should ideally be hosted in a data center (like AWS).
- Exposes local machine to internet
- Requires daemon with system access
- Dependent on local internet connection
- Not recommended for production

Deploy applications from local machines using Cloudflare Tunnels.

Use Cases:

- Development testing
- Home lab applications
- CCTV monitoring systems
- Personal projects

Setup:

```
# Install Cloudflare daemon
curl -L
https://github.com/cloudflare/cloudflared/releases/latest/download/cloudfla
red-linux-amd64.deb -o cloudflared.deb
sudo dpkg -i cloudflared.deb

# Create tunnel
cloudflared tunnel create my-app
cloudflared tunnel route dns my-app app.example.com

# Start tunnel
cloudflared tunnel run my-app
```

Cloudflare Tunnels

Tunnels allow a public-facing URL/IP (which points to a Cloudflare/EC2 server) to **tunnel the request to a local machine**.

- Local machines lack a public IP necessary to expose processes on the internet.

- Tunnels bypass the difficulty of "punching through a router" to get a public IP for HTTPS (Port 443) traffic.

Tunnel Creation Steps (Cloudflare Zero Trust):

1. Go to the Cloudflare dashboard, click **Zero Trust**, then **Networks / Tunnels**.
2. **Install and Authenticate Cloudflare Daemon (Process):** This process runs on the machine that will be tunneled to.
   - *Security Note:* Running the daemon gives Cloudflare access to the machine, so caution is advised.
3. **Create Tunnel:** Select **Cloudflare Daemon**.
4. **Configuration:** Specify the Base URL (e.g., `next-3.10kdev.com`) and the Process URL (e.g., `http://localhost:3000`) where the application is running locally.
   - *Note:* The domain must be managed within Cloudflare for this setup.

Result: Any request to the public URL is routed to the local process (or, in the demonstration, the EC2 machine where the daemon was run). If the local process is stopped, the tunnel reports "bad gateway".

3.6. Deployment Method 7: Cloud-Specific Autoscaling (AWS Elastic Beanstalk/ASG)

Self-hosting provides control, but scaling is poor. Platform as a Service (PaaS) and Serverless scale well, but pricing is high and unpredictable (especially during traffic spikes/attacks), and a premium is paid for external scaling management.

Goal: Self-host applications while retaining a level of **autoscaling**.

Autoscaling Groups (ASG) in AWS

Big cloud providers offer constructs to auto-scale machines.

- In AWS, this is called an **Autoscaling Group** (ASG).
- **Functionality:** Define limits (Min/Max number of servers) and conditions (e.g., if CPU percentage > 50%, increase servers; if CPU < 10%, decrease servers).
- ASGs existed even before container orchestration engines (like Kubernetes).
- **Drawback:** The time it takes to boot up a new server can be high, potentially failing to handle traffic spikes immediately (e.g., for the first 5 minutes).

Deployment Options for ASG

1. **Build from Scratch:** Requires five steps (starting instance, creating an image, creating a launch template, defining a start script, creating ASG with a load balancer).
2. **Elastic Beanstalk (EBS):** AWS service that hides the complexity and performs the five steps under the hood. EBS requires a ZIP file of the Node.js/Next.js code.

Elastic Beanstalk Deployment Steps:

1. **Create Environment:** Search for Elastic Beanstalk and click **Create Environment**. Select **Web Server Environment** (for internet-facing application).
2. **Configuration:** Select the platform (e.g., **Node.js**).
3. **Service Access:** Requires creating and selecting an **IAM role** with necessary permissions.

4. **Instance Profile:** This is a crucial step; if skipped, the environment will crash. An Instance Profile must be created and attached.
5. **Autoscaling Configuration:** Select **Load Balanced Autoscaling Group**. Set minimum (e.g., 1) and maximum (e.g., 5 or 3) machines.
6. **Scaling Strategy:** Based on CPU utilization for easy testing, ideally 70-80% threshold, but 20% is used for demonstration.
7. **Instance Type:** Select a suitable size (e.g., **T3 medium** or C5 large) as smaller types might not be sufficient to build a Next.js application on the server.

Deploying Custom Code to EBS

1. **Prepare Start Script:** The Next.js application must be zipped. The start script within the code handles `npm install`, `npm run build`, and then `next start` on the port passed by Elastic Beanstalk as an environment variable (`PORT`).
   - *Note:* Building locally and only passing the `dist` folder is the ideal approach, but full build steps are included in the archive for robustness due to past errors.
2. **Create Archive:** Zip the complete application folder (e.g., using `zip -r archive-name.zip .` on Mac or right-click compress).
3. **Upload and Deploy:** In the EBS dashboard, upload the ZIP file and click **Deploy**.

Result: Elastic Beanstalk runs the start script on the underlying servers, deploys the application, and sets up a load balancer. The code resides on the machine at `/var/app/current`.

Testing Autoscaling

1. **Check ASG Configuration:** Verify the ASG shows the minimum and maximum capacity set (e.g., Min 1, Max 3).
2. **Set Scaling Policy:** Define a target tracking policy based on average CPU utilization (e.g., scale up if CPU > 20%).
3. **Simulate Load:** Artificially spike CPU usage by running an infinite loop script (`index.js`) on the running EC2 instance.
4. **Observe Scaling:** After a few minutes, the **desired capacity automatically increases** (e.g., from 1 to 3), triggering the start of new instances.
   - Activity logs show the monitor alarm triggered the policy change.

Autoscaling Group Summary:

- ASGs are the easiest way to create autoscaling servers from first principles if raw Node.js applications are used without containerization.
- GCP has a similar construct, but DigitalOcean previously only offered scaling via Kubernetes clusters.

Deploy backend on VMs with automatic scaling capabilities.

AWS Elastic Beanstalk Approach:

1. Prepare Application

```
```

```
{
  "scripts": {
    "start": "next start -p $PORT"
  }
}
```

2. Create ZIP Archive

```
zip -r application.zip . -x node_modules/\*
```

3. Deploy to Elastic Beanstalk

- Create new environment
- Upload ZIP file
- Configure auto-scaling policies

Auto-Scaling Configuration:

- **Minimum instances**: 1
- **Maximum instances**: 5
- **Scale up trigger**: CPU > 70%
- **Scale down trigger**: CPU < 20%

4. Part 3: Containerization and Advanced Deployment

Containerization provides consistent environments across development and production.

4.1. Containerization (Docker)

Containerization is highly recommended for backends.

Benefits of Containerization:

1. **Security/Blast Radius:** Decreases the blast radius by restricting access to the Docker container.
2. **Reduced Overhead:** Avoids the overhead of manually installing dependencies like Node.js/NVM on every EC2 machine.
3. **Portability:** Allows running the application via a single command (`Docker run XYZ`) regardless of the host OS (Mac, Windows, Linux).

Dockerfile for Next.js

The Dockerfile defines the container image structure:

```
# Base image with Node version 20
FROM node:20
# Working directory
WORKDIR /app
```

```
# Copy packages and install dependencies
COPY package*.json .
RUN npm install
# Copy rest of the code
COPY . .
# Build the application
RUN npm run build
# Expose the application port
EXPOSE 3000
# Command to run when container starts
CMD ["npm", "run", "start"]
```

Note: The `package.json` start script should only contain `next start`.

Building and Running the Docker Image

```
docker build -t username/app-name .
docker push username/app-name
```

1. **Build Image:** Use `docker build -t kintsugidev/landing .`.
2. **Run Locally:** Use `docker run -p 3000:3000 kintsugidev/landing`.

Pushing to Docker Hub

- Docker Hub is used to store container images. Public images are free.
- Push the image: `Docker push kintsugidev/landing`.
- For deployment on Linux environments (like Kubernetes clusters), a Linux image is needed: `docker build --platform=linux/amd64 -t kintsugidev/landing:linux .` and then push the Linux tag.

Benefit of Docker Hub: On every code commit, a fresh image can be pushed to Docker Hub, which various runtimes (EBS, Docker Swarm, Kubernetes, decentralized compute) can pull for deployment.

4.2. Deployment Method 8: Container Orchestration (Kubernetes: Enterprise-grade container orchestration for scalable applications)

Container Orchestration (Kubernetes 101):

- Kubernetes (K8s) is an orchestration engine that manages containers.
- **Developer Input:** The developer/DevOps engineer specifies the desired number of containers (**replicas**) and the image.
- **K8s Function:** K8s figures out which machines (nodes) to deploy the containers on and ensures the specified number of containers are always running (**auto-healing**).

Why Kubernetes?

- **Automatic scaling** based on resource usage
- **Self-healing** - restarts failed containers

- **Load balancing** across multiple replicas
- **Rolling updates** with zero downtime
- **Resource management** and optimization

Key Kubernetes Jargon:

- **Deployment:** An object that specifies the number of replicas and the image needed. A deployment creates a **Replica Set**.
- **Replica Set:** Creates a bunch of **Pods**.
- **Pod:** A container (or group of containers) running an instance of the application.
- **Service:** Required to expose the application over the internet, as pods are not exposed by default. Services can be of type **Load Balancer** or **Cluster IP**.
- **Ingress:** Recommended for certificate management out of the box and hosting multiple applications on a single load balancer.

Kubernetes Manifests:

Deployment (deployment.yaml)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: username/app-name:latest
        ports:
        - containerPort: 3000
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"
```

Service (service.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 3000
  type: LoadBalancer
```

3. Deploy to Kubernetes

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Monitoring and Management:

```
# View deployments
kubectl get deployments

# View pods
kubectl get pods

# View services
kubectl get services

# Check logs
kubectl logs <pod-name>

# Scale application
kubectl scale deployment my-app --replicas=5
```

Benefits:

- **Production-ready** scaling
- **High availability** and fault tolerance
- **Resource optimization**
- **Declarative configuration**
- **Ecosystem integration** (monitoring, CI/CD)

Drawbacks:

- **Complex learning curve**

- **Operational overhead**
- **Resource requirements** for cluster management

Depth: Basic Deployment Steps (Using Vultr K8s Cluster)

1. **Create Kubernetes Cluster:** Create a cluster on a provider (e.g., Vultr, AWS, GCP). Specify region and number of machines/nodes (e.g., Delhi, 2 nodes).
2. **Connect CLI:** Download the cluster configuration and paste it into `~/.kube/config`. Verify connection using `kubectl get nodes`.
3. **Create Deployment:** Write a `deployment.yaml` file defining the desired state (e.g., 3 replicas, image: `kintsugidev/landing:linux`, container port: 3000).
   - Apply deployment:

     ```
     kubectl apply -f deployment.yaml
     ```

   - Verify: `kubectl get deployment`, `kubectl get pods` (shows containers/pods starting and eventually running).
4. **Create Service (Load Balancer Type):** Write a `service.yaml` to expose the pods.
   - Service Metadata targets pods with the label `app: next-landing`.
   - Exposes traffic on Port 80, routing to the application's **Target Port 3000**.
   - Type: **LoadBalancer** (automatically creates a load balancer outside the cluster).
   - Apply service:

     ```
     kubectl apply -f service.yaml
     ```

5. **Access and Domain:** The external IP of the newly created load balancer (visible in the provider dashboard, e.g., Vultr) hosts the website.
6. **Attach Domain:** Add an **A record** in DNS management pointing the domain (e.g., `next-5.kintsugidev.com`) to the load balancer IP.

Kubernetes Advantages:

- Provides **Auto-healing** (e.g., if a pod is deleted, a replacement restarts automatically).
- Provides self-healing systems for reliable deployment.
- Speeds up deployment time, offering a scalable way to deploy in minutes once the cluster is understood.
- Allows for advanced concepts like **GitOps**.

Depth: Production Deployment (Using Ingress and Cluster IP Service)

The final production deployment for `kintsugidev.com` uses a more sophisticated Kubernetes architecture:

1. **Deployment** (create).
2. **Service** (create, type: **Cluster IP**).

3. **Ingress** (update existing Ingress to include the new domain and service).
4. **Certificate** (add, using Cert-Manager kind, secret name `Landing-TLS`).
5. **DNS Record** (Update DNS, requiring an IP-based A record at the root level because CNAME is disallowed).

4.3. Deployment Method 9: Decentralized Compute (Akash Network)

Experimental deployment using distributed computing networks.

Concept:

- Use spare computing power from global network
- Market-based pricing model
- Peer-to-peer infrastructure

Characteristics:

- **Immature technology**
- **Unreliable** for production workloads
- **Lower costs** than traditional cloud
- **Limited support** and documentation

When to Consider:

- Non-critical applications
- Cost-sensitive projects
- Experimental workloads
- Learning purposes

This is considered a **dystopian scenario** and immature.

Concept of Decentralized Compute:

- A market where small providers can rent out their available machine compute.
- It counters the monopoly of large cloud providers (AWS, GCP).
- It offers a **competitive market** to buy/rent servers, often cheaper than conventional cloud services (e.g., computing for less than a dollar a month).
- **Akash Network** is one such network.
- *Decentralization Note:* The network relies on IPs, which governments can track, so it is not completely decentralized or anonymous unless the wallet itself is anonymous.

Akash Network Deployment:

1. **Wallet Requirement:** Requires a **Kepler wallet** (on Cosmos) and some AK/AKT tokens (Akash Network token).
2. **Deployment Spec:** Deployment is configured using a spec file highly similar to a **Kubernetes spec file** (creates a service/deployment, exposes Port 80, targets Port 3000 on the container image).
3. **Image:** The application must be containerized (e.g., `kintsugidev/landing:linux`).
4. **Bidding/Payment:** Requires paying a small amount of AKT tokens (e.g., $1 required for transaction approval).

5. **Accept Bid:** Select the cheapest bid with reasonable uptime (e.g., $0.57/month for a half-core machine) and accept the bid.

Result: The application is hosted on a peculiar URL (e.g., `weird-url.ingress.weird-url.cloud`).

Decentralized Compute Benefits (Key takeaway):

- **Cheap:** Computing can be acquired for significantly less than traditional cloud providers ($5-$10/month minimum elsewhere vs. < $1/month here).
- **Containers are Essential:** Containers enable platforms like Akash because the image runs independently with its own file system and dependencies, simplifying the deployment process immensely ("This is the image you need to start").

Market Maturity and Friction: The system is fairly immature and slow. Friction is high (requires specific wallet, specific token).

4.4. Deployment Method 10: Open-Source PaaS/Vercel Alternative (Coolify)

Coolify is pitched as an **open-source Vercel alternative**.

Features:

- **Git integration** like Vercel
  - Includes deployments and **direct integration with Git/GitHub**.
- **Docker-based** deployments
- **Multiple frameworks** support
- **Self-hosted** on your infrastructure
  - It is **self-hosted**, meaning it can be deployed on the user's own infrastructure, making it cheaper than Vercel.
- **Cost-effective** compared to commercial PaaS
- **Vercel-like experience** at lower cost
  - Provides the benefits of Vercel (UI for deployments, tracking status).
- **Full control** over infrastructure
- **No vendor lock-in**
- **Community-driven** development

Self-Hosting:

- To install Coolify, a specific command must be run on a server.
- The system requires connecting to the user's compute (e.g., AWS EC2) via **SSH keys** (e.g., placing the Coolify key in `~/.ssh/authorized_keys`). Coolify then SSHs into the server to deploy applications.

Setup Process:

1. Deploy Coolify on VPS
2. Connect Git repositories
3. Configure deployment settings
4. Automatic deployments on git push

Best Practices

Security:

- Always use **SSL certificates** (HTTPS)
- Implement **proper authentication**
- Regular **security updates**
- **Firewall configuration** for VMs
- **Environment variable** management for secrets

Performance:

- **Enable compression** (gzip/brotli)
- **Optimize images** and assets
- **Implement caching** strategies
- **Use CDN** for static assets
- **Monitor application** performance

Cost Optimization:

- **Right-size resources** based on actual usage
- **Use reserved instances** for predictable workloads
- **Implement auto-scaling** to match demand
- **Monitor billing** and set up alerts
- **Choose appropriate storage** tiers

Development Workflow:

- **Automate deployments** with CI/CD
- **Use staging environments** for testing
- **Implement health checks**
- **Monitor application** logs and metrics
- **Plan disaster recovery** strategies

> As Coolify is Open-Source, feel free to have a look at it's code, To Understand How Deployment Services PaaS Works. https://github.com/coollabsio/coolify

Conclusion

The choice of deployment method depends on:

- **Application type** (static vs dynamic)
- **Expected traffic** patterns
- **Budget constraints**
- **Team expertise**
- **Scalability requirements**
- **Maintenance capacity**

Recommended Learning Path:

1. Start with **Vercel** for learning
2. Progress to **VM deployment** for understanding

3. Master **CDN + Object Store** for static sites
4. Learn **Kubernetes** for enterprise applications
5. Explore **serverless** for cost optimization

Each method has its place in the deployment ecosystem. Understanding multiple approaches enables you to choose the right tool for each specific use case and project requirements.
