Deployment Tools — Topper-Style Notes

- **Why it matters**: Turn code into running apps reliably (local → cloud).
- **Core pieces**: Markdown/docs, assets/images, static hosting, serverless, CI/CD, containers, tunnels, auth, CORS, REST, frameworks.

Markdown (Docs)

- **Use**: Project docs, READMEs, course notes.
- **Essentials**:
  - Headings `#`, lists `-` / `1.`, links `[a](url)`, images `![alt](src)`
  - Fenced code blocks with language: `python,` bash
  - Tables (GFM), task lists `- [ ]` / `- [x]`, blockquotes `>`
  - **GitHub Flavored Markdown**: Strikethrough `~~text~~`, task lists, tables
- **Tools**: `markdownlint`, `pandoc`, VS Code Markdown All in One, `markdown2`.
- **Mini table**: | Task | Tool | Command | | — | — | — | | Lint | markdownlint | npx markdownlint . | | Convert MD→HTML | pandoc | pandoc README.md -o index.html | | Python library | markdown2 | `import markdown2` |

Image Compression

- **Why**: Cut load time and storage.
- **Rules**: Prefer WebP/AVIF; lossless for diagrams; lossy for photos; resize to display size.

Image formats and compression:

- **SVG**: Vector graphics, scales without quality loss
- **WebP**: Modern standard, supports both lossy and lossless
- **PNG**: Lossless compression, good for diagrams
- **JPEG**: Lossy compression, good for photos

Python compression example:

```
 from pathlib import Path
from PIL import Image

async def compress_image(input_path: Path, output_path: Path, quality: int
    """Compress an image while maintaining reasonable quality."""
    with Image.open(input_path) as img:
        # Convert RGBA to RGB if needed
        if img.mode == 'RGBA':
            img = img.convert('RGB')
        # Optimize for web
        img.save(output_path, 'WEBP', quality=quality, optimize=True)
```

Command line tools:

```
 # Convert to WebP
cwebp -q 85 input.png -o output.webp

# Optimize PNG
pngquant --quality=65-80 image.png

# Optimize JPEG
jpegoptim --strip-all --all-progressive --max=85 image.jpg

# Batch convert
mogrify -format webp -quality 85 *.jpg
```

Tools: squoosh.app, ImageOptim, sharp, Pillow

GitHub Pages (Static hosting)

- **What**: Free static hosting from a repo branch.
- **Happy path**:

```
 mkdir my-site && cd my-site && git init
echo "<h1>My Site</h1>" > index.html
git add . && git commit -m "feat(pages): initial commit" && git push origin
# Enable Pages: Settings → Pages → Build from main branch
```

- **Best practices**:
  - Keep site small; optimize images (SVG/WebP/8-bit PNG), preload critical CSS
  - Avoid large binaries; if needed, use Git LFS

- **Related tools**: GitHub Desktop, GitHub CLI, Actions for automation

Google Colab (Notebooks)

- **Pros**: Free GPU/TPU, sharing; **Cons**: session timeouts.
- **Tip**: Mount Drive for persistence; export `.ipynb`/HTML.

Key features:

- Free access to GPUs and TPUs
- Easy sharing of code and execution results
- 12-hour timeout on free tier
- Inconsistent GPU access on free tier

Common operations:

```
 # Mount Google Drive for persistence
from google.colab import drive
drive.mount('/content/drive')

# Install packages
!pip install package_name

# Export notebook
# File → Download → Download .ipynb
# File → Download → Download .html
```

Best practices:

- Mount Google Drive for persistent storage
- Manage dependencies with `!pip install` commands
- Export results before session timeout
- Use GPU/TPU for ML projects when available

Vercel (Serverless)

- **Why**: Simple serverless + static hosting with Git integration.
- **Patterns**:
    - You deploy functions; platform scales them per-request.
    - No arbitrary filesystem access; rely on APIs and `requirements.txt`.

Common use cases:

- Contact forms that email you (runs 2-3 seconds, few times per day)
- Photo conversion tools (runs 5-10 seconds per upload)
- Chatbots answering business questions (runs 1-2 seconds per question)
- Newsletter sign-ups (runs 1 second per sign-up)
- Webhooks posting sales to Discord (runs 1 second per sale)

FastAPI quickstart:

```
# requirements.txt
fastapi
```

```
# main.py
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"message": "Hello, World!"}
```

```
// vercel.json
{
  "builds": [{ "src": "main.py", "use": "@vercel/python" }],
  "routes": [{ "src": "/(.*)", "dest": "main.py" }]
}
```

```
npx vercel           # preview deploy
npx vercel --prod    # production deploy
npx vercel env add   # add environment variables
```

- **In code**: `os.environ.get('SECRET_KEY')`
- **Videos**: Product walkthrough; Deploy FastAPI on Vercel

GitHub Actions (CI/CD)

- **Use**: Automate tests, builds, deploys, data updates.
- **Concepts**: workflows in `.github/workflows/*.yml`, jobs, steps, `uses`, secrets, runners, caching.

Key concepts:

- **Workflows**: YAML configuration files in `.github/workflows/`
- **Triggers**: push, PR, schedule, manual dispatch
- **Jobs**: Units of work that run on runners
- **Steps**: Individual tasks within jobs
- **Actions**: Reusable units of work from marketplace
- **Secrets**: Secure environment variables
- **Caching**: Speed up workflows by caching dependencies

Sample (daily job writing ISS position to a JSONL):

```
 name: Log ISS Location Data Daily
on:
  schedule:
    - cron: "0 12 * * *"
  workflow_dispatch:
jobs:
  collect-iss-data:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v5
      - name: Fetch ISS location data
        run: |
          uv run --with requests python << 'EOF'
          import requests
          data = requests.get('http://api.open-notify.org/iss-now.json').te
          with open('iss-location.jsonl', 'a') as f:
              f.write(data + '\n')
          EOF
      - name: Commit and push changes
        run: |
          git config --local user.email "github-actions[bot]@users.noreply.
          git config --local user.name "github-actions[bot]"
          git add iss-location.jsonl
          git commit -m "Update ISS position data [skip ci]" || exit 0
          git push
```

- **Tips**: Use secrets for tokens; cache deps; monitor free tier limits.

Containers: Docker / Podman

- **Why**: Reproducible runtime; ship code + deps as an image.

Docker vs Podman:

- **Docker**: Industry standard, widely used
- **Podman**: Compatible with Docker, better security, more open license
- **Recommendation**: Use Podman but Docker works the same way

Podman init:

```
podman machine init && podman machine start
```

- **Common ops**:

```
podman pull python:3.11-slim
podman run -it python:3.11-slim
podman ps -a && podman stop <id> && podman rm <id>
podman build -t py-hello . && podman run -it py-hello
podman login docker.io && podman push py-hello:latest docker.io/$DOCKER_HUB
```

- **Dockerfile mini**:

```
FROM python:3.11-slim
WORKDIR /app
RUN echo 'print("Hello, world!")' > app.py
CMD ["python", "app.py"]
```

- **Security/Tools**: `podman scan`, Trivy, Dive, Skopeo

ngrok (Tunnels)

- **What**: Expose localhost for webhooks/demos.

Setup:

```
# Get authtoken from dashboard
ngrok config add-authtoken $YOUR_AUTHTOKEN

# Start HTTP tunnel
uvx ngrok http 8000
```

Useful features:

- `ngrok http file://.` - serve local files
- `--response-header-add "Access-Control-Allow-Origin: *"` - enable CORS
- `--oauth google` - restrict access with Google Auth
- `--ua-filter-deny ".*bot$"` - reject bot user agents

Use cases:

- Testing webhooks
- Sharing work in progress
- Debugging applications in production-like environments

CORS

- **Rule**: Set `Access-Control-Allow-Origin` (and methods/headers) to permit frontend origin.

Key concepts:

- **Same-Origin Policy**: Browsers block requests between different origins by default
- **CORS Headers**: Server responses must include specific headers to allow cross-origin requests
- **Preflight Requests**: Browsers send OPTIONS requests to check if the actual request is allowed
- **Credentials**: Special handling required for requests with cookies or authentication

Common CORS headers:

```
 Access-Control-Allow-Origin: https://example.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Allow-Credentials: true
```

FastAPI implementation:

```
 from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()
```

```
app.add_middleware(CORSMiddleware, allow_origins=["*"]) # Allow GET request
# Or, provide more granular control:
app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://example.com"],  # Allow a specific domain
    allow_credentials=True,  # Allow cookies
    allow_methods=["GET", "POST", "PUT", "DELETE"],  # Allow specific metho
    allow_headers=["*"],  # Allow all headers
)
```

Common CORS errors:

- `No 'Access-Control-Allow-Origin' header` : Configure server to send proper CORS headers
- `Request header field not allowed` : Add required headers to `Access-Control-Allow-Headers`
- `Credentials flag` : Set both `credentials: 'include'` and `Access-Control-Allow-Credentials: true`
- `Wild card error` : Cannot use `*` with credentials; specify exact origins

REST APIs

- **Principles**: resources, verbs (GET/POST/PUT/PATCH/DELETE), status codes (2xx,4xx,5xx), idempotency.

HTTP Methods:

- `GET` : Retrieve data
- `POST` : Create new data
- `PUT/PATCH` : Update existing data
- `DELETE` : Remove data

Status Codes:

- `2xx` : Success (200 OK, 201 Created)
- `4xx` : Client errors (400 Bad Request, 404 Not Found)
- `5xx` : Server errors (500 Internal Server Error)

Best Practices:

1. **Use Nouns for Resources**: `/users` , `/posts` (not `/getUsers`)

2. **Version Your API**: `/api/v1/users`, `/api/v2/users`
3. **Handle Errors Consistently**: Standard error response format
4. **Use Query Parameters for Filtering**: `/api/posts?status=published&category=tech`
5. **Implement Pagination**: `/api/posts?page=2&limit=10`

Tools: Postman, Swagger/OpenAPI, HTTPie, JSON Schema

FastAPI (Web framework)

- **Strengths**: Type hints, Pydantic models, async, auto docs.

Key features:

- Modern Python web framework for building APIs
- Automatic interactive documentation
- Fast, easy to use, production-ready
- Built-in type hints and validation

Snippet:

```
 from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
class Item(BaseModel):
    name: str
    price: float
@app.post("/items")
def create_item(item: Item):
    return {"ok": True, "item": item}
```

Google Auth

- **Use**: OAuth2 login; secure endpoints; manage tokens and refresh.

Why Google Auth:

- Most commonly implemented single sign-on mechanism
- Popular and user-friendly (users log in with existing Google accounts)
- Secure: supports OAuth2 and OpenID Connect

Setup process:

1. Go to Google Cloud Console – Credentials
2. Click **Create Credentials > OAuth client ID**
3. Choose **Web application**, set authorized redirect URIs
4. Copy **Client ID** and **Client Secret** to `.env` file

Environment variables:

```
 GOOGLE_CLIENT_ID=your-client-id.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=your-client-secret
```

Use cases:

- Allow access to specific users only
- Fetch user's personalized information
- Display different content based on user
- Secure API endpoints

GitHub Codespaces (Remote Dev Environments)

- **What**: Cloud-hosted development environment built into GitHub.
- **Why**: Reproducible onboarding, anywhere access, rapid experimentation.

Key benefits:

- **Reproducible onboarding**: Say goodbye to "works on my machine" woes
- **Anywhere access**: Jump back into your project from any device
- **Rapid experimentation**: Spin up short-lived environments on any branch/commit/PR

Quick setup:

```
 # Via GitHub CLI
gh auth login
gh codespace create --repo OWNER/REPO
gh codespace list    # List all codespaces
gh codespace code    # opens in your local VS Code
gh codespace ssh     # SSH into the codespace
```

Features to explore:

- **Dev Containers**: Set up environment using `devcontainer.json` or Dockerfile
- **Prebuilds**: Build complex repos in advance for faster startup
- **Port Forwarding**: Automatic port detection and forwarding
- **Secrets & Variables**: Safe environment variables in repo settings
- **Dotfiles Integration**: Customize shell settings and tools
- **Machine Types**: Pick from VMs with 2 to 32 cores
- **VS Code & CLI Integration**: Browser VS Code and desktop editor
- **GitHub Actions**: Power prebuilds and CI/CD inside codespaces
- **Copilot in Codespaces**: AI suggestions in the editor

Ollama (Local LLMs)

- **What**: Run LLMs locally; pull models; compatible HTTP endpoints.

Key features:

- **Model management**: `list` / `pull` — Install and switch among Llama 3.3, DeepSeek-R1, Gemma 3, Mistral, Phi-4, and more
- **Local inference**: `run` — Execute prompts entirely on-device for privacy and zero latency
- **Persistent server**: `serve` — Expose a local REST API for multi-session chats
- **Version pinning**: `pull model:tag` — Pin exact model versions for reproducible demos
- **Resource control**: `--threads` / `--context` — Tune CPU/GPU usage and context window

Basic usage:

```
 # List installed and available models
ollama list

# Download/pin a specific model version
ollama pull gemma3:1b-it-qat

# Run a one-off prompt
ollama run gemma3:1b-it-qat 'Write a haiku about data visualization'
```

```
# Launch a persistent HTTP API on port 11434
ollama serve

# Interact programmatically over HTTP
curl -X POST http://localhost:11434/api/chat \
     -H 'Content-Type: application/json' \
     -d '{"model":"gemma3:1b-it-qat","prompt":"Hello, world!"}'
```

Real-world use cases:

- **Quick prototyping**: Brainstorm slide decks or blog outlines offline
- **Data privacy**: Summarize sensitive documents on-device
- **CI/CD integration**: Validate PR descriptions or test YAML configurations
- **Local app embedding**: Power desktop or web apps via local REST API

Comparison Table

Tool	Role	Best for	Notes
GitHub Pages	Static hosting	Docs, small sites	Free; branch-based
Vercel	Static + serverless	Jamstack apps	Git-integrated; envs
Actions	CI/CD automation	Tests/deploys	Secrets, caching
Docker/Podman	Runtime packaging	Repro builds	Security scans

Checks and tips

- CORS: if it works in curl but fails in browser, inspect preflight response headers.
- Prefer idempotent updates (PUT) where feasible; reserve POST for create/trigger.
- Trim container size with multi-stage builds, pinned bases, non-root users.
- Pages: after DNS change, allow cert issuance time; enforce HTTPS in settings.

Advanced theory and tricky exam asks

- **CORS preflight**: Triggered by non-simple methods/headers; browser sends OPTIONS; server must reply with matching `Access-Control-Allow-*` headers including `Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers`.
- **REST idempotency**: GET, PUT, DELETE are idempotent; POST is not; PATCH is not guaranteed.
- **HTTP caching**: `Cache-Control`, `ETag` / `If-None-Match`, `Last-Modified` / `If-Modified-Since` —avoid double-caching with CDN + browser.
- **CI caching**: Key strategy includes OS, lockfile hash, and tool versions to avoid stale caches; scope caches per job when needed.
- **Container layering**: Put frequently-changing files later; pin base images; avoid root; mount read-only FS in prod.
- **Supply-chain security**: Scan images (Trivy), pin registries/tags; minimal base images (distroless/alpine with care).
- **Vercel cold starts**: Usually small; mitigate by warmup hits or edge functions; design for idempotent retries.
- **GitHub Pages DNS**: `CNAME` ownership; TTL and propagation; HTTPS cert issuance may lag after DNS changes.

Possible exam questions:

- Why does a preflight fail even though the API works in curl?
- Show an idempotent update endpoint and explain when POST is appropriate.
- How to cut container size and reduce attack surface while keeping functionality?

Deep dive details

CORS preflight example:

```
 OPTIONS /api/items HTTP/1.1
Origin: https://app.example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization, Content-Type
```

Server must reply:

```
 HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.example.com
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers: Authorization, Content-Type
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
```

REST idempotency matrix:

Method	Idempotent	Typical use
GET	✔️	Read resource
PUT	✔️	Replace resource
DELETE	✔️	Remove resource
PATCH	➖	Partial update (may not be idempotent)
POST	❌	Create/trigger action

CI cache example (YAML):

```
- uses: actions/cache@v4
  with:
    path: ~/.cache/uv
    key: ${{ runner.os }}-uv-${{ hashFiles('uv.lock') }}
```

Container layering best practices:

- Put `COPY . .` after installing deps to leverage cache.
- Pin base images; avoid `latest`.
- Run as non-root; use read-only FS in prod.
- Multi-stage builds: build → slim runtime image.

GitHub Pages DNS checklist:

- Add `CNAME` file with custom domain.
- Point DNS CNAME to `username.github.io` (or A/AAAA for apex via GitHub IPs).
- Wait for TLS certificate; force HTTPS in settings.

Related topics – quick notes

CORS (cross-origin requests)

- Purpose: allow a browser app on one origin to call an API on another.
- Why: browsers enforce same-origin; APIs must opt-in via headers.
- Core: simple vs preflighted; OPTIONS check; allow-origin/methods/headers; Vary for caches.

```
 OPTIONS /api/items
Origin: https://app.example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization, Content-Type
```

```
 HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.example.com
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers: Authorization, Content-Type
Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers
```

- Pitfalls: wildcard origin with credentials; missing Vary; caching stale preflights.
- Checklist: list methods/headers; echo allowed origin (or tight list); include Vary.

CI/CD workflows

- Purpose: automate tests, builds, deploys, data refresh jobs.
- Why: reliable, repeatable delivery.
- Core: triggers (push, PR, schedule), jobs, runners, secrets, caching.

```
- uses: actions/cache@v4
  with:
    path: ~/.cache/uv
    key: ${{ runner.os }}-uv-${{ hashFiles('uv.lock') }}
```

- Pitfalls: overbroad cache keys; leaking secrets; long-lived runners with stale state.
- Checklist: pin actions; minimal perms; secret scanning; cache on lockfiles.

Static sites

- Purpose: publish HTML/CSS/JS without servers.
- Why: cheap, fast, secure.
- Core: small assets, image optimization, custom domain via CNAME + DNS, HTTPS.
- Pitfalls: committing large binaries; forgetting to enforce HTTPS; DNS propagation delays.
- Checklist: minify assets; WebP/AVIF images; add CNAME; verify TLS.

OAuth2 basics

- Purpose: authorize apps to act on user’s behalf.
- Why: scoped access; token rotation.
- Core: auth code + PKCE for browsers/mobile; client credentials for server-to-server; refresh vs access tokens.
- Pitfalls: storing refresh tokens insecurely; mixing flows; missing scopes.
- Checklist: choose flow; store secrets safely; rotate; least-privilege scopes.

Serverless hosting

- Purpose: deploy functions without managing servers.
- Why: scale-to-zero, per-request billing.
- Core: edge vs serverless runtime; routing config; env vars per environment; no local FS assumptions.

```
{
  "builds": [{ "src": "main.py", "use": "@vercel/python" }],
  "routes": [{ "src": "/(.*)", "dest": "main.py" }]
}
```

- Pitfalls: cold starts; timeouts; ephemeral storage.
- Checklist: set timeouts; idempotent handlers; use object stores for files.

Containers

- Purpose: package code + deps into reproducible images.

- Why: consistent runs across machines.
- Core: multi-stage builds, non-root users, pinned bases, scans.

```
 FROM python:3.11-slim AS base
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
USER 1000:1000
CMD ["python","app.py"]
```

- Pitfalls: `latest` tags; root users; bloated layers.
- Checklist: pin tags; drop root; scan; slim images.

Tunnels

- Purpose: expose local servers for webhooks/demos.
- Why: quick external access.
- Core: auth tokens; URL whitelisting; request signing.
- Pitfalls: leaving tunnels open; unauthenticated webhooks.
- Checklist: rotate tokens; verify signatures/IPs; time-box exposure.

Notebooks in the cloud

- Purpose: run notebooks without local setup.
- Why: GPUs, sharing.
- Core: session limits; persistent storage mounts; artifact export.
- Pitfalls: data loss on timeout; hidden runtime differences.
- Checklist: mount drive; checkpoint outputs; pin package versions.

Remote dev environments

- Purpose: standardized dev setups.
- Why: consistency with CI.
- Core: devcontainer config, extensions, preinstalled deps.
- Pitfalls: drift from CI images; missing secrets.
- Checklist: base on CI image; document env vars; bootstrap tasks.

Slides from Markdown

- Purpose: fast docs → slides.
- Why: single source; easy versioning.
- Core: Markdown sections → slides, themes, export to PDF.
- Pitfalls: too much text; unreadable contrast.
- Checklist: one idea per slide; large fonts; test in projector lighting.