{ PLATFORM ENGINEERING }

Python for DevOps & SRE

A Senior Engineer's Field Manual

Guides · Examples · Scenarios · Interview Prep

From core Python to Kubernetes operators, cloud automation, observability, and 50+ senior interview questions.

Practical Platform Engineering Series

2026 Edition **FREE SAMPLE**

s.mp

author

Contents

Free Sample – Preview Edition

Thank you for previewing **Python for DevOps & SRE: A Senior Engineer's Field Manual**.

This free sample contains the complete front matter, the full 12-week learning plan, the entire first module (*Python Fundamentals for DevOps*, with runnable examples and line-by-line explanations), and the interview quick-reference appendix – so you can judge the depth and teaching style before you buy.

The full edition (200+ pages) additionally includes:

- Seven more in-depth modules: Infrastructure Automation, CI/CD, Monitoring & Observability, Cloud Automation, Kubernetes Operations, Security & Compliance, and SRE Scenarios.
- 50+ senior-level interview questions with model answers, plus five deep-dive system-design scenarios.
- Progressive exercises with fully worked solutions.
- A downloadable companion-code bundle – every script in the book, ready to run.

Copyright & Disclaimer

Python for DevOps & SRE: A Senior Engineer's Field Manual

Disclaimer: The code, patterns, and practices in this book are provided for educational purposes. While every effort has been made to ensure accuracy, the author and publisher assume no responsibility for errors, omissions, or damages resulting from the use of the information contained herein. Always test automation in a non-production environment before deploying to production systems. Cloud provider APIs, pricing, and service behavior change over time; verify against current official documentation.

Preface

Most Python books teach you the language. This book teaches you how to **wield Python as a platform engineer** – someone who keeps production systems reliable, automates away toil, and responds calmly at 3 A.M. when the pager fires.

If you are a DevOps engineer, Site Reliability Engineer (SRE), or platform engineer aiming for a senior or lead role, the code you write is different from that of a web developer. Your scripts run unattended in CI/CD pipelines, cron jobs, and Kubernetes pods. They must be **idempotent**, **observable**, and **fail gracefully**. A missing `try/except` or a forgotten `--dry-run` flag is not a bug – it is an outage.

This field manual is organized as a **12-week learning path** followed by a comprehensive **interview preparation** section. Every chapter follows the same rhythm:

- **Guide** – the concepts and why they matter for operations work
- **Examples** – complete, runnable code you can adapt to your environment
- **Scenarios** – real production situations and how to automate them
- **Exercises** – progressive challenges that build on each other

The final chapters contain over fifty senior-level interview questions with model answers, five deep-dive system-design scenarios, and a quick-reference appendix of patterns and numbers every SRE should know by heart.

Who This Book Is For

- Engineers moving from junior/mid to **senior or lead** DevOps/SRE roles
- Developers transitioning into platform or infrastructure engineering
- Anyone preparing for a **senior DevOps/SRE interview**
- Teams standardizing on Python for automation and tooling

What You Need

- Python 3.10 or newer
- Comfort with the Linux command line
- Familiarity with at least one cloud provider
- Docker installed locally
- Curiosity and a willingness to break things (in staging)

How to Use This Book

Work through the modules in order – they build on one another. Read the guide in the morning, run the examples at midday, and attempt a scenario in the evening. Type the code yourself rather than copying it; muscle memory matters in an incident. When you reach the interview section, practice explaining your reasoning **out loud** – senior interviews reward clear thinking over memorized syntax.

Let's begin.

Part I – The Learning Path

Python for DevOps/SRE - Complete Learning Plan

Overview

This plan is structured for a Senior/Lead DevOps-SRE engineer who needs production-grade Python skills. Focus is on **practical automation**, **reliability engineering**, and **infrastructure-as-code** – not academic CS.

Phase 1: Foundation (Week 1-2)

Goal: Write clean, production-ready Python scripts

Topic	Why It Matters for DevOps
Data structures (dict, list, set)	Config parsing, inventory management
File I/O and pathlib	Log parsing, config generation
subprocess & os modules	Shell command orchestration
Error handling patterns	Graceful failure in automation
Type hints & dataclasses	Self-documenting infra code
Context managers	Resource cleanup (connections, files)
Generators & iterators	Memory-efficient log processing
Decorators	Retry logic, caching, timing

Phase 2: Networking & APIs (Week 3-4)

Goal: Interact with any service programmatically

Topic	Why It Matters for DevOps
requests/httpx	REST API automation
asyncio & aiohttp	Parallel infra operations
paramiko/fabric	SSH automation
socket programming	Health checks, port scanning

Topic	Why It Matters for DevOps
API client design	Reusable service integrations
Rate limiting & retries	Production-safe API calls
Webhook handlers (Flask/FastAPI)	Event-driven automation

Phase 3: Infrastructure Automation (Week 5-6)

Goal: Automate infrastructure provisioning and config

Topic	Why It Matters for DevOps
Jinja2 templating	Config generation at scale
YAML/JSON/TOML processing	IaC file manipulation
Terraform wrapper scripts	Plan validation, drift detection
Ansible modules in Python	Custom automation
Docker SDK	Container lifecycle management
Git automation (GitPython)	GitOps workflows

Phase 4: Cloud & Kubernetes (Week 7-8)

Goal: Automate cloud infrastructure and K8s operations

Topic	Why It Matters for DevOps
Boto3 (AWS)	EC2, S3, IAM, Lambda automation
google-cloud-python	GCP resource management
Kubernetes Python client	Pod management, CRD handling
Custom K8s operators (kopf)	Automated reconciliation
Pulumi (Python)	Infrastructure-as-real-code

Phase 5: Monitoring & Observability (Week 9-10)

Goal: Build monitoring solutions and automate incident response

Topic	Why It Matters for DevOps
prometheus_client	Custom metrics exporters
Log aggregation scripts	Structured logging pipelines
Alert routing logic	PagerDuty/Slack integration
SLI/SLO calculators	Error budget tracking

Topic	Why It Matters for DevOps
Trace correlation	Distributed system debugging

Phase 6: Security & Reliability (Week 11-12)

Goal: Harden systems and build resilient automation

Topic	Why It Matters for DevOps
Secrets management	Vault/AWS SM integration
Certificate automation	TLS rotation scripts
Policy-as-code (OPA)	Compliance checking
Chaos engineering	Fault injection tooling
Incident runbooks as code	Automated remediation
Capacity planning scripts	Resource forecasting

Daily Practice Routine

```
Morning (30 min):  Read one module's guide.md
Midday (45 min):   Work through the examples/
Evening (30 min):  Attempt one scenario challenge
```

Milestones & Checkpoints

- ☐ **Week 2**: Can write a CLI tool that parses configs and manages services
- ☐ **Week 4**: Can build an API client with retries, auth, and error handling
- ☐ **Week 6**: Can generate Terraform/Ansible configs programmatically
- ☐ **Week 8**: Can manage K8s resources and write a simple operator
- ☐ **Week 10**: Can build a custom Prometheus exporter
- ☐ **Week 12**: Can write an automated incident response runbook

---

Key Principles for DevOps Python

1. **Idempotency** – Scripts must be safe to run multiple times
2. **Observability** – Always log what you're doing and why
3. **Fail gracefully** – Never leave infrastructure in a broken state
4. **Dry-run mode** – Always support `--dry-run` for destructive ops
5. **Configuration over code** – Externalize environment-specific values
6. **Testing** – Test your automation before it touches production

Module 1: Python Fundamentals for DevOps/SRE

Why This Matters

As a DevOps/SRE lead, you write Python differently than a web developer. Your code: - Runs in CI/CD pipelines, cron jobs, and Kubernetes pods - Must be idempotent (safe to re-run) - Handles infrastructure state and failures gracefully - Often runs unattended – logging and error handling are critical

---

1.1 Data Structures for Infrastructure

Infrastructure is mostly nested configuration plus set-based comparisons, so mastering Python's core data structures lets you model and reconcile system state cleanly. These examples show the dictionary and set patterns you will reach for constantly when managing services and servers.

Dictionaries – Your Config Swiss Army Knife

```
# Representing infrastructure state
service_config = {
    "name": "payment-api",
    "replicas": 3,
    "resources": {
        "cpu": "500m",
        "memory": "256Mi"
    },
    "env_vars": {
        "DB_HOST": "postgres.internal",
        "CACHE_TTL": "300"
    },
    "health_check": {
        "path": "/healthz",
        "interval": 30,
        "timeout": 5
    }
}

# Deep merge configs (common pattern for overlays)
import copy

def deep_merge(base: dict, override: dict) -> dict:
    """Merge override into base, handling nested dicts (non-mutating)."""
    result = copy.deepcopy(base)
```

```
    for key, value in override.items():
        if key in result and isinstance(result[key], dict) and isinstance(value, dict):
            result[key] = deep_merge(result[key], value)
        else:
            result[key] = value
    return result

# Usage: environment-specific overrides
prod_override = {"replicas": 5, "resources": {"memory": "512Mi"}}
prod_config = deep_merge(service_config, prod_override)
```

How it works.

- `service_config` models a full service as nested dicts (`resources`, `env_vars`, `health_check`), mirroring how tools like Kubernetes and Terraform represent state.
- `deep_merge` recursively walks matching nested dict keys, so an overlay like `prod_override` changes only what it specifies and leaves the rest of `base` intact – exactly how environment overlays work.
- Starting from `copy.deepcopy(base)` means the merged result never aliases the original's nested branches, so mutating the result can never mutate the source config – the function is safe to re-run (idempotent). A shallow `base.copy()` would leave untouched nested dicts shared with the original, a subtle source of bugs.

Sets – Fast Membership Testing

```
# Find servers that need patching
all_servers = {"web-1", "web-2", "web-3", "db-1", "db-2"}
patched_servers = {"web-1", "db-1"}
needs_patching = all_servers - patched_servers
# {'web-2', 'web-3', 'db-2'}

# Find common security groups between two instances
sg_instance_a = {"sg-web", "sg-monitoring", "sg-ssh"}
sg_instance_b = {"sg-api", "sg-monitoring", "sg-ssh"}
shared_groups = sg_instance_a & sg_instance_b
# {'sg-monitoring', 'sg-ssh'}
```

How it works.

- The difference operator `all_servers - patched_servers` returns the servers still needing patches in a single expression, with no loops.
- The intersection operator `sg_instance_a & sg_instance_b` finds security groups shared by two instances, a common audit and compliance check.
- Sets give O(1) membership tests, so these comparisons stay fast even across thousands of hosts.

1.2 File Operations with pathlib

File and config handling is the backbone of most automation, and `pathlib` gives you an object-oriented, cross-platform way to do it safely. This `ConfigManager` shows the read, write, discover, and backup operations you need when juggling configs across environments.

```
from pathlib import Path
import json
import yaml

class ConfigManager:
    """Manage application configs across environments."""

    def __init__(self, config_dir: str = "/etc/myapp"):
        self.config_dir = Path(config_dir)

    def load_yaml(self, filename: str) -> dict:
        config_path = self.config_dir / filename
        if not config_path.exists():
            raise FileNotFoundError(f"Config not found: {config_path}")
        return yaml.safe_load(config_path.read_text())

    def write_yaml(self, filename: str, data: dict) -> None:
        config_path = self.config_dir / filename
        config_path.parent.mkdir(parents=True, exist_ok=True)
        config_path.write_text(yaml.dump(data, default_flow_style=False))

    def find_all_configs(self, pattern: str = "*.yaml") -> list[Path]:
        return sorted(self.config_dir.rglob(pattern))

    def backup_config(self, filename: str) -> Path:
        from datetime import datetime
        source = self.config_dir / filename
        backup_dir = self.config_dir / "backups"
        backup_dir.mkdir(exist_ok=True)
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        dest = backup_dir / f"{source.stem}_{timestamp}{source.suffix}"
        dest.write_text(source.read_text())
        return dest
```

How it works.

- The constructor stores `config_dir` as a `Path`, so joins like `self.config_dir / filename` work on any OS without manual string handling.
- `write_yaml` calls `parent.mkdir(parents=True, exist_ok=True)` to create directories on demand, keeping the write idempotent.
- `find_all_configs` uses `rglob` to recursively discover files by pattern, while `backup_config` stamps a timestamp into the filename so backups never overwrite each other.

1.3 subprocess – Running System Commands Safely

Automation constantly shells out to system tools, and doing that safely – with timeouts, logging, and a dry-run mode – is what separates production scripts from fragile ones. This `run_command` wrapper is a reusable, defensive pattern for every command you execute.

```
import subprocess
import shlex
import logging

logger = logging.getLogger(__name__)


def run_command(
    cmd: str | list[str],
    timeout: int = 60,
    dry_run: bool = False,
    capture: bool = True
) -> subprocess.CompletedProcess:
    """
    Execute a shell command safely.

    Args:
        cmd: Command string or list of args
        timeout: Max seconds to wait
        dry_run: If True, only log what would run
        capture: If True, capture stdout/stderr
    """
    if isinstance(cmd, str):
        cmd_list = shlex.split(cmd)
    else:
        cmd_list = cmd

    logger.info(f"Running: {' '.join(cmd_list)}")

    if dry_run:
        logger.info("[DRY RUN] Skipping execution")
        return subprocess.CompletedProcess(cmd_list, 0, "", "")

    try:
        result = subprocess.run(
            cmd_list,
            capture_output=capture,
            text=True,
            timeout=timeout,
            check=True  # Raises CalledProcessError on non-zero exit
        )
        return result
    except subprocess.TimeoutExpired:
        logger.error(f"Command timed out after {timeout}s: {cmd_list}")
        raise
```

```
    except subprocess.CalledProcessError as e:
        logger.error(f"Command failed (exit {e.returncode}): {e.stderr}")
        raise


# Real usage: Check disk space across servers
def check_disk_usage(threshold: int = 80) -> list[dict]:
    """Find partitions above threshold% usage."""
    result = run_command("df -h --output=pcent,target")
    alerts = []
    for line in result.stdout.strip().split("\n")[1:]:  # Skip header
        parts = line.strip().split()
        if len(parts) == 2:
            usage = int(parts[0].rstrip("%"))
            mount = parts[1]
            if usage > threshold:
                alerts.append({"mount": mount, "usage": usage})
    return alerts
```

How it works.

- `shlex.split` turns a command string into an argument list, avoiding the shell-injection risk of `shell=True`.
- The `dry_run` flag logs the command and returns early, letting you preview destructive operations before running them for real.
- `check=True` raises `CalledProcessError` on non-zero exits, and the `try/except` blocks log timeouts and failures with context before re-raising – essential for unattended jobs.
- `check_disk_usage` shows a concrete application: parsing `df` output to alert on partitions above a threshold.

1.4 Error Handling Patterns for Automation

Networks and APIs fail transiently, so robust automation must retry intelligently and stop hammering broken dependencies. This section pairs a retry decorator with a circuit breaker – two resilience patterns every SRE relies on.

```
import time
import functools
import logging
from typing import Callable, TypeVar, Any

T = TypeVar("T")
logger = logging.getLogger(__name__)


def retry(
    max_attempts: int = 3,
    delay: float = 1.0,
```

```
    backoff: float = 2.0,
    exceptions: tuple = (Exception,)
) -> Callable:
    """
    Retry decorator with exponential backoff.
    Essential for any network/API operations in DevOps scripts.
    """
    def decorator(func: Callable[..., T]) -> Callable[..., T]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> T:
            current_delay = delay
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions as e:
                    if attempt == max_attempts:
                        logger.error(
                            f"{func.__name__} failed after {max_attempts} attempts: {e}"
                        )
                        raise
                    logger.warning(
                        f"{func.__name__} attempt {attempt}/{max_attempts} "
                        f"failed: {e}. Retrying in {current_delay:.1f}s..."
                    )
                    time.sleep(current_delay)
                    current_delay *= backoff
        return wrapper
    return decorator


# Usage
@retry(max_attempts=5, delay=2.0, exceptions=(ConnectionError, TimeoutError))
def fetch_service_status(url: str) -> dict:
    """Fetch service health with automatic retries."""
    import httpx
    response = httpx.get(url, timeout=10)
    response.raise_for_status()
    return response.json()


# Circuit breaker pattern -- prevent cascading failures
class CircuitBreaker:
    """Stop calling a failing service to prevent cascade."""

    def __init__(self, failure_threshold: int = 5, reset_timeout: int = 60):
        self.failure_threshold = failure_threshold
        self.reset_timeout = reset_timeout
        self.failures = 0
        self.last_failure_time = 0
        self.state = "closed"  # closed=normal, open=blocked, half-open=testing
```

```
    def call(self, func: Callable, *args, **kwargs) -> Any:
        if self.state == "open":
            if time.time() - self.last_failure_time > self.reset_timeout:
                self.state = "half-open"
            else:
                raise RuntimeError(f"Circuit is OPEN -- service unavailable")

        try:
            result = func(*args, **kwargs)
            if self.state == "half-open":
                self.state = "closed"
                self.failures = 0
            return result
        except Exception as e:
            self.failures += 1
            self.last_failure_time = time.time()
            if self.failures >= self.failure_threshold:
                self.state = "open"
                logger.critical(f"Circuit OPENED after {self.failures} failures")
            raise
```

How it works.

- The `retry` decorator wraps a function and re-runs it up to `max_attempts`, multiplying the wait by `backoff` each time for exponential backoff.
- `functools.wraps` preserves the wrapped function's name and metadata, so logs and tracebacks stay meaningful.
- The `exceptions` tuple lets callers retry only specific errors (like `ConnectionError`), so genuine bugs are not silently retried.
- `CircuitBreaker` counts failures and flips to an `open` state once `failure_threshold` is hit, blocking calls until `reset_timeout` passes to prevent cascading failures.

1.5 CLI Tools with Click

Good DevOps tooling ships as ergonomic CLIs, and Click plus Rich give you argument parsing, safety prompts, and readable output with little boilerplate. This example builds a service-management CLI with `deploy` and `status` commands.

```
"""
Example: A DevOps CLI tool for service management.
Run: python service_cli.py deploy --env production --service payment-api
"""
import click
from rich.console import Console
from rich.table import Table

console = Console()
```

```python
@click.group()
@click.option("--verbose", "-v", is_flag=True, help="Enable debug logging")
@click.option("--dry-run", is_flag=True, help="Show what would happen")
@click.pass_context
def cli(ctx, verbose, dry_run):
    """DevOps Service Management CLI."""
    ctx.ensure_object(dict)
    ctx.obj["verbose"] = verbose
    ctx.obj["dry_run"] = dry_run


@cli.command()
@click.option("--env", type=click.Choice(["dev", "staging", "production"]), required=True)
@click.option("--service", required=True, help="Service name to deploy")
@click.option("--version", default="latest", help="Image version/tag")
@click.pass_context
def deploy(ctx, env, service, version):
    """Deploy a service to the specified environment."""
    dry_run = ctx.obj["dry_run"]

    console.print(f"[bold blue]Deploying[/] {service}:{version} to {env}")

    if env == "production" and not dry_run:
        if not click.confirm("[WARN]  Deploy to PRODUCTION?"):
            raise click.Abort()

    # Deployment logic here
    steps = [
        f"Pull image {service}:{version}",
        f"Run pre-deploy checks",
        f"Update {env} deployment",
        f"Wait for rollout",
        f"Run smoke tests",
    ]

    for step in steps:
        if dry_run:
            console.print(f"  [dim][DRY RUN][/dim] {step}")
        else:
            console.print(f"  [green][OK][/green] {step}")


@cli.command()
@click.option("--env", type=click.Choice(["dev", "staging", "production"]), required=True)
def status(env):
    """Show status of all services in an environment."""
    table = Table(title=f"Services in {env}")
    table.add_column("Service", style="cyan")
    table.add_column("Status", style="green")
    table.add_column("Replicas")
```

```
    table.add_column("Version")

    # Would fetch real data from K8s/cloud API
    services = [
        ("payment-api", "Running", "3/3", "v2.1.0"),
        ("user-service", "Running", "2/2", "v1.8.3"),
        ("notification", "Degraded", "1/2", "v3.0.1"),
    ]

    for name, state, replicas, version in services:
        status_style = "green" if state == "Running" else "red"
        table.add_row(name, f"[{status_style}]{state}[/]", replicas, version)

    console.print(table)


if __name__ == "__main__":
    cli()
```

How it works.

- The `@click.group` with `pass_context` shares flags like `verbose` and `dry_run` across every subcommand via `ctx.obj`.
- `deploy` uses `click.Choice` to constrain `env` to valid values and `click.confirm` to require explicit approval before production changes.
- The `status` command builds a Rich `Table` to render service state as a colored, aligned report instead of raw text.

1.6 Structured Logging

Unattended tools are only as debuggable as their logs, and structured JSON logging is what makes them searchable in aggregation systems like ELK or Loki. This section shows a custom formatter and a reusable logging setup.

```
"""Production-grade logging setup for DevOps tools."""
import logging
import json
import sys
from datetime import datetime, timezone


class JSONFormatter(logging.Formatter):
    """Output logs as JSON for log aggregation systems."""

    def format(self, record: logging.LogRecord) -> str:
        log_entry = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "level": record.levelname,
            "message": record.getMessage(),
```

```
            "logger": record.name,
            "module": record.module,
            "function": record.funcName,
        }

        # Add extra fields (correlation IDs, service info, etc.)
        if hasattr(record, "extra_fields"):
            log_entry.update(record.extra_fields)

        if record.exc_info:
            log_entry["exception"] = self.formatException(record.exc_info)

        return json.dumps(log_entry)


def setup_logging(
    level: str = "INFO",
    json_output: bool = False,
    service_name: str = "devops-tool"
) -> logging.Logger:
    """Configure logging for production use."""
    logger = logging.getLogger(service_name)
    logger.setLevel(getattr(logging, level.upper()))

    # Idempotent: clear existing handlers so calling this twice does not
    # attach duplicate handlers (which would print every line multiple times).
    logger.handlers.clear()
    logger.propagate = False

    handler = logging.StreamHandler(sys.stdout)

    if json_output:
        handler.setFormatter(JSONFormatter())
    else:
        handler.setFormatter(logging.Formatter(
            "%(asctime)s [%(levelname)s] %(name)s: %(message)s"
        ))

    logger.addHandler(handler)
    return logger


# Usage
logger = setup_logging(json_output=True, service_name="deploy-tool")
logger.info("Deployment started", extra={"extra_fields": {
    "service": "payment-api",
    "environment": "production",
    "deploy_id": "dep-abc123"
}})
```

How it works.

- `JSONFormatter` overrides `format` to emit each record as a JSON object with a UTC timestamp, level, and source location that log platforms can index directly.
- It merges optional `extra_fields` (correlation IDs, service names) and attaches formatted exceptions whenever `exc_info` is present.
- `setup_logging` wires the formatter to a stdout handler and toggles between JSON and human-readable output, so the same tool works locally and in production.
- It clears any existing handlers first (`logger.handlers.clear()`) so calling it more than once is safe – otherwise each call would add another handler and every log line would print two, three, or more times.

1.7 Type Hints & Dataclasses for Infrastructure Models

Modeling infrastructure with typed dataclasses catches mistakes early and turns loose dicts into validated objects. This `ServiceDefinition` shows how to represent a deployable service and generate a Kubernetes-style spec from it.

```
from dataclasses import dataclass, field
from enum import Enum
from typing import Optional


class Environment(str, Enum):
    DEV = "dev"
    STAGING = "staging"
    PRODUCTION = "production"


class ServiceState(str, Enum):
    RUNNING = "running"
    STOPPED = "stopped"
    DEGRADED = "degraded"
    DEPLOYING = "deploying"


@dataclass
class ResourceLimits:
    cpu: str = "250m"
    memory: str = "256Mi"

    def to_k8s_format(self) -> dict:
        return {"cpu": self.cpu, "memory": self.memory}


@dataclass
class ServiceDefinition:
    name: str
    image: str
    environment: Environment
    replicas: int = 1
```

```
    resources: ResourceLimits = field(default_factory=ResourceLimits)
    env_vars: dict[str, str] = field(default_factory=dict)
    labels: dict[str, str] = field(default_factory=dict)
    port: int = 8080
    health_check_path: str = "/healthz"

    def __post_init__(self):
        """Validate after initialization."""
        if self.replicas < 1:
            raise ValueError(f"Replicas must be >= 1, got {self.replicas}")
        if self.environment == Environment.PRODUCTION and self.replicas < 2:
            raise ValueError("Production services need at least 2 replicas")

    @property
    def full_image(self) -> str:
        return self.image if ":" in self.image else f"{self.image}:latest"

    def to_deployment_spec(self) -> dict:
        """Generate K8s-like deployment spec."""
        return {
            "apiVersion": "apps/v1",
            "kind": "Deployment",
            "metadata": {
                "name": self.name,
                "labels": {**self.labels, "app": self.name},
            },
            "spec": {
                "replicas": self.replicas,
                "template": {
                    "spec": {
                        "containers": [{
                            "name": self.name,
                            "image": self.full_image,
                            "ports": [{"containerPort": self.port}],
                            "resources": {"limits": self.resources.to_k8s_format()},
                            "env": [
                                {"name": k, "value": v}
                                for k, v in self.env_vars.items()
                            ],
                        }]
                    }
                }
            }
        }


# Usage
svc = ServiceDefinition(
    name="payment-api",
    image="registry.internal/payment-api:v2.1.0",
    environment=Environment.PRODUCTION,
```

```
    replicas=3,
    resources=ResourceLimits(cpu="500m", memory="512Mi"),
    env_vars={"DB_HOST": "postgres.internal", "LOG_LEVEL": "info"},
)
```

How it works.

- Enums like `Environment` and `ServiceState` restrict fields to valid values, so an invalid environment cannot be constructed.
- `field(default_factory=...)` gives each instance its own `ResourceLimits` and dict defaults, avoiding shared-mutable-default bugs.
- `__post_init__` validates invariants (`replicas >= 1`, production needs at least 2), failing fast on bad configuration.
- `to_deployment_spec` renders the typed model into a K8s Deployment dict, bridging your Python objects and real cluster manifests.

Exercises

1. **Config Merger**: Write a script that merges base + environment YAML configs
2. **Log Parser**: Parse nginx access logs, find top 10 IPs and 5xx errors
3. **Health Checker**: CLI tool that checks multiple endpoints in parallel
4. **Service Inventory**: Read a YAML inventory, validate it, output a status table
5. **Safe Executor**: Build a `run_command()` wrapper with retry, timeout, and dry-run

Part II – Interview Preparation (Preview)

Quick Reference: Python Patterns for DevOps Interviews

Pattern 1: Idempotent Operations

```
def ensure_resource_exists(name: str, desired_state: dict) -> str:
    """Create or update -- safe to call multiple times."""
    existing = get_resource(name)
    if existing is None:
        return create_resource(name, desired_state)
    elif existing != desired_state:
        return update_resource(name, desired_state)
    else:
        return "no-change"  # Already in desired state
```

Pattern 2: Graceful Shutdown

```
import signal
import sys

class GracefulShutdown:
    def __init__(self):
        self.should_stop = False
        signal.signal(signal.SIGTERM, self._handler)
        signal.signal(signal.SIGINT, self._handler)

    def _handler(self, signum, frame):
        self.should_stop = True

    # Usage in a worker loop:
    # while not shutdown.should_stop:
    #     process_next_item()
    # cleanup()
```

Pattern 3: Builder Pattern for Complex Objects

```
class DeploymentBuilder:
    def __init__(self, name: str):
        self._spec = {"name": name}

    def with_replicas(self, n: int):
        self._spec["replicas"] = n; return self
```

```
    def with_image(self, image: str):
        self._spec["image"] = image; return self

    def with_resources(self, cpu: str, memory: str):
        self._spec["resources"] = {"cpu": cpu, "memory": memory}; return self

    def build(self) -> dict:
        return self._spec

# deploy = DeploymentBuilder("api").with_replicas(3).with_image("v2").build()
```

Pattern 4: Strategy Pattern for Multi-Cloud

```
from abc import ABC, abstractmethod

class CloudProvider(ABC):
    @abstractmethod
    def create_instance(self, spec: dict) -> str: ...

    @abstractmethod
    def terminate_instance(self, instance_id: str) -> None: ...

class AWSProvider(CloudProvider):
    def create_instance(self, spec): ...
    def terminate_instance(self, id): ...

class GCPProvider(CloudProvider):
    def create_instance(self, spec): ...
    def terminate_instance(self, id): ...

# Usage: provider = get_provider(config["cloud"])  # Returns correct impl
```

Pattern 5: Observer/Event Pattern

```
from typing import Callable

class EventBus:
    def __init__(self):
        self._handlers: dict[str, list[Callable]] = {}

    def subscribe(self, event: str, handler: Callable):
        self._handlers.setdefault(event, []).append(handler)

    def emit(self, event: str, data: dict):
        for handler in self._handlers.get(event, []):
            handler(data)
```

```
# bus.subscribe("deploy.started", notify_slack)
# bus.subscribe("deploy.failed", trigger_rollback)
# bus.emit("deploy.started", {"service": "api", "version": "v2"})
```

Pattern 6: Pagination Iterator

```
def paginate_api(client, method: str, **kwargs):
    """Generic paginator for any API with token-based pagination."""
    while True:
        response = getattr(client, method)(**kwargs)
        yield from response.get("items", [])

        next_token = response.get("next_token")
        if not next_token:
            break
        kwargs["page_token"] = next_token

# for instance in paginate_api(ec2, "describe_instances"):
#     process(instance)
```

Pattern 7: Configuration with Validation (Pydantic)

```
from pydantic import BaseModel, field_validator

class ServiceConfig(BaseModel):
    name: str
    port: int
    replicas: int
    image: str

    @field_validator("replicas")
    @classmethod
    def validate_replicas(cls, v):
        if v < 1: raise ValueError("Need at least 1 replica")
        return v

    @field_validator("image")
    @classmethod
    def validate_image(cls, v):
        if ":latest" in v: raise ValueError("No :latest in production")
        return v
```

Pattern 8: Command Pattern for Undo/Redo

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self) -> None: ...

    @abstractmethod
    def undo(self) -> None: ...

class ScaleDeployment(Command):
    def __init__(self, name: str, target: int):
        self.name = name
        self.target = target
        self.previous = None

    def execute(self):
        self.previous = get_current_replicas(self.name)
        set_replicas(self.name, self.target)

    def undo(self):
        if self.previous is not None:
            set_replicas(self.name, self.previous)
```

Key Numbers to Know

Metric	Value	Context
99.9% SLO monthly downtime	43.2 minutes	"three nines"
99.99% SLO monthly downtime	4.3 minutes	"four nines"
99.999% SLO monthly downtime	26 seconds	"five nines"
TCP connection timeout	3-way handshake ~1 RTT	SYN -> SYN-ACK -> ACK
DNS TTL typical	60-300 seconds	Affects failover speed
K8s pod startup	5-30 seconds	Image pull + init containers
EBS snapshot time	Minutes for incremental	First snapshot slower
RDS failover (Multi-AZ)	60-120 seconds	Automatic
Route53 health check interval	10 or 30 seconds	Configurable
ALB draining timeout	300 seconds default	Connection draining

Common Gotchas to Mention

1. **Mutable default arguments**: `def f(items=[])` – shared across calls!
2. **Late binding closures**: `lambdas in loops` capture variable, not value
3. **String concatenation in loops**: Use `"".join()` for O(n) vs $O(n^2)$

4. **`except Exception` vs bare `except:`**: `except Exception` does NOT catch `KeyboardInterrupt` or `SystemExit` – those subclass `BaseException`, not `Exception`. Only a bare `except:` (or `except BaseException`) swallows them, which is why bare except is dangerous.
5. **`is` vs `==` for None**: Always use `x is None`, never `x == None`
6. **Import circular deps**: Import inside function or restructure
7. **`datetime.now()` in tests**: Use dependency injection or freezegun
8. **subprocess shell=True**: Security risk (injection), avoid it

Get the Full Edition

You have reached the end of the free sample.

If this preview was useful, the complete book turns it into a full field manual you will keep coming back to.

The full edition includes:

- **8 in-depth modules** – from core Python to Kubernetes operators, cloud automation, observability, security, and SRE scenarios.
- **50+ senior interview questions** with model answers, five system-design scenarios, and a quick-reference cheat sheet.
- **Progressive exercises with worked solutions** so you can practice, then check your work.
- **A companion-code bundle** – every script in the book, organized by chapter and ready to run.

Format: instant-download PDF and DOCX, syntax-highlighted code, full table of contents.

Thank you for reading. – s.mp, Practical Platform Engineering Series