TECHNICAL WHITEPAPER

Deploying Python APIs on Fly.io

Django REST Framework & FastAPI with PostgreSQL and Redis

A practical guide for developers and DevOps engineers seeking production-grade cloud deployments without the operational overhead of traditional cloud platforms.

Prepared by:

H4 Tech SOLUTIONS

www.h4tech.com

July 2026

EDITION	AUDIENCE	STACK
2026 v1.0	Developers & DevOps Engineers	Django DRF · FastAPI · Postgres · Redis

Table of Contents

Executive Summary

Modern software teams deploying Python REST APIs face a fundamental tension: cloud platforms like AWS and Google Cloud Platform offer robust infrastructure with global reach, but introduce significant operational complexity before a single line of application code can run in production. This whitepaper examines Fly.io as an alternative deployment target for Django REST Framework and FastAPI applications, analyzes the tradeoffs against traditional cloud platforms, and provides production-ready configuration patterns for PostgreSQL and Redis.

The core thesis is straightforward: for the majority of Python API workloads — internal tooling, SaaS backends, mobile app APIs, and data services operating at startup-to-mid-scale — the infrastructure overhead imposed by AWS ECS, RDS, and related services does not deliver proportional value. Fly.io compresses that overhead to a single configuration file and a handful of CLI commands while retaining meaningful operational control.

> **Key Finding**
> A production-grade Django DRF or FastAPI deployment on Fly.io — including PostgreSQL and Redis — can be achieved in under 15 minutes. An equivalent setup on AWS ECS with RDS typically requires 2–6 hours and significantly more ongoing maintenance.

This document is structured as both a conceptual guide and a hands-on reference. Readers will find platform comparisons, architecture diagrams, annotated configuration files, migration strategies, scaling patterns, and a curated list of operational best practices and common pitfalls.

1. Introduction & Motivation

The proliferation of cloud-native tooling over the past decade has produced two distinct developer experiences. Large engineering organizations have invested in platform teams that abstract infrastructure complexity behind internal deployment platforms. Smaller teams — startups, indie developers, and growing product companies — have had to become infrastructure generalists to deploy even modest applications.

Deploying a Python REST API on AWS in 2025 requires decisions and configuration across roughly twelve service boundaries: IAM roles, VPC configuration, subnet groups, security groups, ECR image repositories, ECS task definitions, ECS services, Application Load Balancers, Target Groups, RDS parameter groups, Secrets Manager entries, and CloudWatch log groups. Many of these services have interdependencies that must be resolved in the correct order. The cognitive overhead is substantial, and it grows non-linearly with the number of services involved.

Fly.io represents a different design philosophy: compress the configuration surface to a single file (fly.toml), automate the networking between services, and provide first-class CLI primitives that map to human-meaningful actions (fly deploy, fly postgres attach, fly secrets set). The platform is built on Firecracker micro-VMs and WireGuard networking — the same foundational technologies used by AWS Lambda and modern zero-trust networks — but exposed through an interface designed for application developers rather than infrastructure engineers.

This whitepaper does not advocate for Fly.io as a universal replacement for AWS or GCP. There are workloads, compliance requirements, and scale characteristics where traditional cloud platforms are the correct choice. What it argues is that the default choice of AWS for a new Python API project often carries significant hidden costs in time, complexity, and ongoing maintenance that are not reflected in the compute bill.

> **Scope**
> This document covers Django REST Framework and FastAPI deployments specifically. The patterns and tradeoffs discussed are applicable to other Python frameworks (Flask, Litestar, etc.) but the configuration examples use DRF and FastAPI conventions.

2. Platform Overview: What Is Fly.io?

Fly.io is a cloud platform that runs application containers as Firecracker micro-VMs across a network of globally distributed edge data centers. The technical stack distinguishes it from both traditional PaaS platforms and managed Kubernetes offerings.

2.1 Firecracker Micro-VMs

Unlike container runtimes such as Docker or containerd that share a host kernel, Fly runs each container inside a Firecracker micro-VM — a lightweight virtual machine with its own kernel, developed by AWS and used internally for Lambda and Fargate. This provides stronger isolation guarantees than container namespacing while maintaining sub-second boot times. For Python API developers, the practical effect is that a VM misbehaving (exhausting memory, entering a crash loop) cannot affect neighboring VMs on the same host.

2.2 Private WireGuard Networking

Every application deployed to Fly.io within an organization is connected via a WireGuard mesh network. Applications communicate using private DNS names following the pattern [app-name].internal. A Django application connecting to a Fly Postgres cluster does so at my-db.internal:5432 — no VPC peering configuration, no NAT gateway, no security group rules. The network is encrypted by default and the routing is automatic.

This single architectural decision eliminates the largest source of AWS configuration complexity for most Python API deployments: the VPC setup required to allow private communication between ECS tasks, RDS instances, and ElastiCache clusters.

2.3 The fly.toml Contract

All application configuration lives in a single TOML file (fly.toml) that is committed to the project repository. This file specifies the Docker build configuration, VM sizing, environment variables, health check parameters, deployment strategy, HTTP service routing, and process group definitions. The file is the complete infrastructure-as-code representation for a Fly application, and changes to it are version-controlled alongside the application code.

2.4 Anycast Edge Proxy

Fly operates an anycast proxy layer that handles TLS termination, HTTP/2 multiplexing, and request routing to the nearest healthy VM. TLS certificates are provisioned automatically for the fly.dev subdomain and for custom domains added via fly certs. There is no load balancer to configure, no ACM certificate request to approve, and no listener rule to define.

- Fly.io free tier: 3 shared-CPU VMs, 3GB persistent storage, 160GB outbound transfer per month
- VM options range from shared-cpu-1x (256MB RAM) to performance-8x (16GB RAM, 8 dedicated CPUs)
- 35+ global regions including US, Europe, Asia-Pacific, and South America
- Fly Postgres uses Patroni for automatic failover — no manual intervention for primary/replica promotion

3. Platform Comparison: Fly.io vs AWS vs GCP

The following comparison uses a representative scenario: deploying a Python REST API container with a PostgreSQL database, a Redis cache, and basic autoscaling. The comparison evaluates setup time, operational commands, cost, and flexibility.

Task	Fly.io	AWS (ECS/RDS)	GCP (Cloud Run)
Initial setup	**~10 minutes**	2–6 hours (VPC, IAM, ALB)	1–3 hours
Deploy container	**fly deploy**	ECR → task revision → service update	gcloud run deploy
Add PostgreSQL	**fly postgres create**	VPC subnets + SG + parameter group	Cloud SQL wizard (5+ steps)
Set secrets	**fly secrets set KEY=val**	Secrets Manager + IAM policy	Secret Manager + permissions
Scale to 3 VMs	**fly scale count 3**	ASG + Launch Template + ALB group	Cloud Run min/max config
Multi-region	**fly regions add nrt**	Global Accelerator + Route53	Cloud Run multi-region
SSH into VM	**fly ssh console**	SSM Session Manager or bastion	gcloud compute ssh
View live logs	**fly logs**	CloudWatch (laggy, $$$)	Cloud Logging
Cost (small API)	**Free – $5/mo**	$50–120/mo minimum	$10–40/mo
Vendor lock-in	**Low (Docker-native)**	High (ECS, Fargate, proprietary)	Medium

3.1 Cost Analysis

The cost difference between Fly.io and AWS for small-to-medium Python API workloads is substantial and is frequently underestimated by developers new to AWS. The minimum viable AWS setup for a production API includes costs that have no Fly equivalent:

- **NAT Gateway:** $32–45/month for a single AZ, required for ECS tasks in private subnets to reach the internet
- **Application Load Balancer:** $16–25/month minimum for the ALB itself, plus $0.008 per LCU
- **RDS db.t3.micro:** $25–30/month for the smallest viable Postgres instance
- **CloudWatch Logs:** $0.50/GB ingestion, $0.03/GB storage — moderate APIs generate significant log volume

These line items create an effective floor of $70–100/month for a minimal AWS production setup before any compute costs. Fly.io's equivalent — one VM + Fly Postgres + Upstash Redis — starts at approximately $5–15/month and scales proportionally with actual usage.

3.2 When AWS or GCP Remains the Right Choice

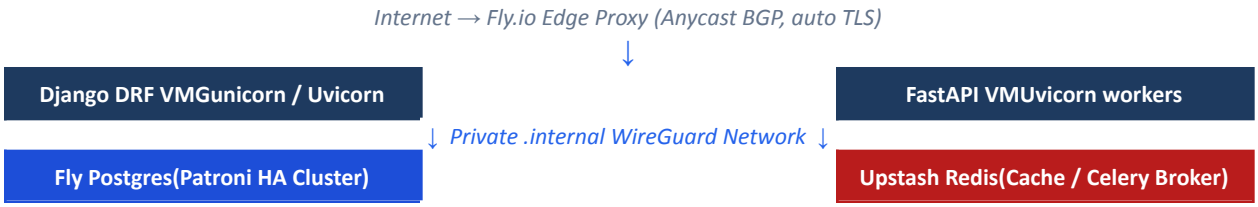
This comparison should not be read as a universal recommendation to avoid traditional cloud platforms. Specific conditions make AWS or GCP the appropriate choice:

- **Compliance requirements:** HIPAA Business Associate Agreements, FedRAMP authorization, PCI-DSS Level 1 — AWS has more mature certification coverage
- **Existing infrastructure investment:** Organizations with existing VPCs, IAM structures, and AWS service integrations have legitimate reasons to standardize on AWS
- **Large ML/GPU workloads:** AWS SageMaker, Bedrock, and GPU instance types have no Fly equivalent at scale
- **Enterprise IAM requirements:** Fine-grained attribute-based access control, AWS Organizations, and Control Tower have no Fly equivalent
- **Dedicated DevOps team:** Organizations with platform engineering teams can manage AWS complexity and extract value from its breadth

4. Core Architecture & Concepts

4.1 Reference Architecture

The following diagram illustrates the standard architecture for a Python API deployment on Fly.io, showing the relationship between the edge proxy, application VMs, and backend services over the private network.



4.2 Fly Primitives

Understanding the core Fly.io primitives and their equivalents in traditional cloud platforms reduces the learning curve for developers migrating from AWS or GCP.

Fly Concept	AWS/GCP Equivalent	What It Does
`App`	ECS Service / Cloud Run Service	A named unit containing VMs, domains, secrets, and routing
`Machine`	EC2 instance / Fargate task	A Firecracker micro-VM running a container image
`fly.toml`	task-definition.json + service config	All application config in one file, committed to git
`Volume`	EBS volume	Persistent disk attached to a specific machine
`Secrets`	Secrets Manager / Parameter Store	Encrypted env vars, injected at VM startup
`Proxy`	Application Load Balancer	Global anycast proxy with automatic TLS
`.internal DNS`	VPC private DNS	WireGuard mesh names — e.g. my-db.internal:5432

Private Networking Advantage

All Fly apps within an organization communicate over an encrypted WireGuard mesh. A Django app connects to Postgres at my-db.internal:5432 with zero configuration. This eliminates VPC peering, security group rules, and NAT gateway costs — the largest source of AWS networking complexity for API deployments.

5. Deploying Django REST Framework

5.1 Dockerfile

The Dockerfile follows a layer-caching-optimized pattern: system dependencies, then Python packages, then application code. This order ensures that code changes (the most frequent operation) do not invalidate the pip install layer.

Dockerfile

```
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PORT=8000

WORKDIR /app

# System deps for psycopg2
RUN apt-get update && apt-get install -y \
    libpq-dev gcc --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

# Install deps before copying app code — preserves Docker cache
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
RUN python manage.py collectstatic --noinput

RUN useradd -m -u 1000 appuser && USER appuser

EXPOSE 8000
CMD ["gunicorn", "myproject.wsgi:application",
     "--bind", "0.0.0.0:8000", "--workers", "2", "--timeout", "120"]
```

5.2 fly.toml Configuration

The fly.toml file is the complete infrastructure definition for a Django application on Fly.io. Notable sections include the release_command for zero-downtime migrations and the http_service concurrency settings for autoscaling.

fly.toml

```
app = "my-django-api"
```

```
primary_region = "lax"

[build]
  dockerfile = "Dockerfile"

[env]
  # Non-sensitive config only — secrets use fly secrets set
  DJANGO_SETTINGS_MODULE = "myproject.settings.production"

[deploy]
  # Runs BEFORE traffic switches — safe zero-downtime migrations
  release_command = "python manage.py migrate --noinput"

[http_service]
  internal_port = 8000
  force_https = true
  auto_stop_machines = true
  auto_start_machines = true
  min_machines_running = 1   # keep 1 warm — eliminates cold starts

  [http_service.concurrency]
    type = "requests"
    soft_limit = 150
    hard_limit = 200

[[vm]]
  memory = "512mb"
  cpu_kind = "shared"
  cpus = 1

[checks]
  [checks.health]
    grace_period = "10s"
    interval = "30s"
    method = "GET"
    path = "/api/health/"
    port = 8000
    timeout = "5s"
    type = "http"
```

5.3 Production Settings

Django production settings on Fly.io follow the twelve-factor app pattern: all configuration is read from environment variables. Fly injects DATABASE_URL automatically when a Postgres cluster is attached via fly postgres attach.

```
settings/production.py
```

```
from .base import *
import dj_database_url

DEBUG = False
SECRET_KEY = os.environ["SECRET_KEY"]
ALLOWED_HOSTS = os.environ.get("ALLOWED_HOSTS", "").split(",")

# Fly injects DATABASE_URL via fly postgres attach
DATABASES = {
    "default": dj_database_url.config(
        default=os.environ["DATABASE_URL"],
        conn_max_age=600,         # persistent connections, 10 min
        conn_health_checks=True,  # validate before reuse (Django 4.1+)
        ssl_require=True,
        options={"options": "-c statement_timeout=30000"},
    )
}

CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": os.environ["REDIS_URL"],
        "OPTIONS": {"CLIENT_CLASS": "django_redis.client.DefaultClient"},
    }
}

CELERY_BROKER_URL = os.environ["REDIS_URL"]

# Whitenoise serves static files without a separate CDN
MIDDLEWARE = ["whitenoise.middleware.WhiteNoiseMiddleware"] + list(MIDDLEWARE)
```

5.4 Celery Worker Process Groups

Background task workers (Celery) can run as a separate process group within the same Fly application, sharing secrets and private network access without requiring a separate app deployment.

```
fly.toml — process groups
[processes]
  web    = "gunicorn myproject.wsgi --bind 0.0.0.0:8000 -w 2"
  worker = "celery -A myproject worker -l info -c 2"
  beat   = "celery -A myproject beat -l info"

# Scale each process group independently
# fly scale count web=2 worker=3 beat=1
```

6. Deploying FastAPI

6.1 Async-First Configuration

FastAPI's async architecture requires specific uvicorn configuration for correct operation behind Fly's anycast proxy. Two flags are mandatory: --proxy-headers instructs uvicorn to trust forwarded headers, and --forwarded-allow-ips "*" allows Fly's proxy IPs to forward client IP information.

```
Dockerfile
FROM python:3.12-slim

ENV PYTHONUNBUFFERED=1 PORT=8000
WORKDIR /app

RUN apt-get update && apt-get install -y libpq-dev gcc --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

RUN useradd -m -u 1000 appuser && USER appuser
EXPOSE 8000

# --proxy-headers: required for correct client IP behind Fly proxy
# --forwarded-allow-ips "*": trust Fly forwarded headers
# Never use --reload in production
CMD ["uvicorn", "app.main:app",
     "--host", "0.0.0.0", "--port", "8000",
     "--workers", "2",
     "--proxy-headers", "--forwarded-allow-ips", "*"]
```

6.2 Async SQLAlchemy Database Layer

FastAPI applications should use asyncpg as the Postgres driver for non-blocking database I/O. The DATABASE_URL injected by Fly uses the postgresql:// scheme and must be converted to postgresql+asyncpg:// for the async engine.

```
app/database.py
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession, async_sessionmaker
import os

# Convert Fly-injected URL to async driver scheme
```

```
DATABASE_URL = os.environ["DATABASE_URL"].replace(
    "postgresql://", "postgresql+asyncpg://"
)

engine = create_async_engine(
    DATABASE_URL,
    pool_size=5,
    max_overflow=10,
    pool_pre_ping=True,     # validate connections before use
    pool_recycle=1800,      # recycle after 30 min
)

AsyncSessionLocal = async_sessionmaker(
    engine, class_=AsyncSession, expire_on_commit=False
)

async def get_db():
    async with AsyncSessionLocal() as session:
        yield session
```

6.3 Application Lifespan & Health

app/main.py

```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from .database import engine

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup: connections are lazy-initialized
    # Migrations run via release_command = "alembic upgrade head"
    yield
    # Shutdown: cleanly close connection pool
    await engine.dispose()

app = FastAPI(title="My API", version="1.0.0", lifespan=lifespan)

@app.get("/health")
async def health():
    return {"status": "ok"}
```

7. PostgreSQL: Provisioning & Operations

7.1 Fly Postgres vs Managed Services

Fly Postgres runs as a Fly application configured with Patroni for high availability and automatic failover. It is a self-managed cluster on infrastructure you control — not a fully managed service like AWS RDS. This distinction has operational implications: OS-level patching, major version upgrades, and storage resizing require manual intervention. For teams that cannot afford database downtime or have limited operational capacity, Neon (serverless Postgres) or Supabase provide a fully managed alternative that works seamlessly with Fly.io via a DATABASE_URL secret.

7.2 Provisioning Commands

```
Terminal
# Create a Postgres cluster
fly postgres create \
  --name my-app-db \
  --region lax \
  --vm-size shared-cpu-1x \
  --volume-size 10

# Attach to app — injects DATABASE_URL secret automatically
# Creates a least-privilege user (not superuser)
fly postgres attach my-app-db --app my-django-api

# Interactive psql session
fly postgres connect -a my-app-db

# Proxy for GUI tools (TablePlus, DBeaver, pgAdmin)
fly proxy 5432 -a my-app-db
```

7.3 Connection Pooling

Django and FastAPI applications with multiple workers can exhaust the default Postgres connection limit of 100. Configure connection pooling at the application level as a first step; for higher-concurrency workloads, PgBouncer or Supabase's built-in pooler should be considered.

```
Connection pool configuration
# Django: dj-database-url with connection reuse
```

```
DATABASES = {
    "default": dj_database_url.config(
        conn_max_age=600,           # reuse connections 10 min
        conn_health_checks=True,    # validate before reuse
        ssl_require=True,
        options={"options": "-c statement_timeout=30000"},
    )
}

# FastAPI: SQLAlchemy async pool settings
engine = create_async_engine(
    DATABASE_URL,
    pool_size=5,
    max_overflow=10,
    pool_pre_ping=True,
)
```

7.4 Safe Migration Patterns

Database migrations in production require careful ordering to avoid table locks and downtime. The following patterns are safe for tables with millions of rows:

- **Adding a nullable column:** `ALTER TABLE orders ADD COLUMN notes TEXT;` — no lock, instant
- **Adding an index:** `CREATE INDEX CONCURRENTLY idx_orders_user ON orders(user_id);` — non-blocking
- **Adding NOT NULL:** Add with DEFAULT first, backfill, then add constraint — never in one ALTER
- **Renaming a column:** Add new column → dual-write → backfill → switch reads → drop old column (across multiple deploys)

8. Redis: Caching & Async Task Queues

8.1 Redis Options on Fly.io

Fly offers two Redis deployment patterns. The choice between them involves a tradeoff between operational simplicity and cost predictability.

	Upstash Redis (Serverless)	Self-Managed Redis VM
Billing	Per request ($0.2/100K)	Per VM/hour
Free tier	10,000 req/day	Uses one of your 3 free VMs
Idle cost	$0 (scales to zero)	VM runs 24/7
Recommended for	Most API workloads	High-throughput / Celery-heavy
Setup	fly redis create	fly launch --image redis:7-alpine

8.2 Provisioning Upstash Redis

```
Terminal
# Create Upstash Redis (recommended)
fly redis create --name my-redis --region lax --no-replicas

# Attach — injects REDIS_URL as a secret
fly redis attach my-redis --app my-django-api

# Verify
fly secrets list -a my-django-api   # → REDIS_URL  set
```

8.3 Security Requirement

> **Never Expose Redis Publicly**
>
> Redis has no built-in authentication model adequate for internet exposure. All Redis VMs on Fly.io must have no public IP allocated. Traffic must stay on the .internal private WireGuard network. Run fly ips list -a my-redis to verify — the output should show no public IPv4 or IPv6 addresses.

9. Secrets & Configuration Management

Fly encrypts secrets at rest using AES-256 and injects them as environment variables at VM startup. This replaces AWS Secrets Manager without requiring IAM policies, SDK calls, or sidecar containers.

9.1 Core Commands

Terminal

```
# Set individual secrets (triggers rolling VM restart)
fly secrets set SECRET_KEY="..." DATABASE_URL="..." -a my-app

# Bulk import from a .env file (never commit .env to git)
fly secrets import < .env.production -a my-app

# List secret names (values are never displayed)
fly secrets list -a my-app

# Remove a secret
fly secrets unset OLD_KEY -a my-app

# Generate Django SECRET_KEY and set in one command
fly secrets set SECRET_KEY=$(python -c \
  "from django.core.management.utils import get_random_secret_key;
print(get_random_secret_key())")
```

9.2 Configuration Hierarchy

✓ DO	✗ DO NOT
Use fly secrets set for all sensitive values: SECRET_KEY, DATABASE_URL, API keys, tokens.	Do not put secrets in fly.toml [env]. This file is committed to git and visible to anyone with repository access.
✓ DO	✗ DO NOT
Add .env, .env.*, and .env.production to .gitignore before your first commit.	Do not log request bodies, authentication tokens, or API keys to stdout. fly logs is visible to all org members.

10. Scaling & Multi-Region Deployment

10.1 Vertical Scaling

```
Terminal
# List available VM sizes
fly platform vm-sizes

# Scale VM size (takes effect immediately)
fly scale vm performance-1x -a my-app

# Or configure in fly.toml
[[vm]]
  memory = "1gb"
  cpu_kind = "performance"
  cpus = 1
```

10.2 Horizontal Scaling

Fly's edge proxy automatically load-balances across all running machines for an application. No load balancer configuration, target group registration, or health check listener rules are required.

```
Terminal
# Scale to 3 machines (load balanced automatically)
fly scale count 3 -a my-app

# Scale process groups independently
fly scale count web=2 worker=3 beat=1 -a my-app
```

10.3 Autoscaling Configuration

```
fly.toml
[http_service]
  auto_stop_machines = true    # stop idle VMs after ~15 minutes
  auto_start_machines = true   # cold-start on incoming traffic
  min_machines_running = 1     # keep 1 warm — no cold starts for users

  [http_service.concurrency]
    type = "requests"
```

```
    soft_limit = 150   # scaling signal threshold
    hard_limit = 200   # reject above this
```

min_machines_running Setting

Set min_machines_running = 0 for background workers and development environments where cold start latency (2–8 seconds) is acceptable. Set to 1 for all user-facing HTTP services in production. The cost of keeping one machine warm is minimal compared to the user experience impact of cold starts.

10.4 Multi-Region Deployment

```
Terminal
# Add regions — VMs provisioned automatically
fly regions add nrt ams lhr -a my-app

# Scale machines per region
fly scale count 2 --region nrt -a my-app

# Add Postgres read replica in Tokyo
fly volumes create pg_data --region nrt --size 20 -a my-app-db

# Fly Postgres routes reads to nearest replica automatically
```

11. Operational Best Practices

11.1 Deployment

✓ DO	✗ DO NOT
Use release_command = "python manage.py migrate --noinput" in fly.toml. Migrations run in a temporary VM before traffic switches — if they fail, the old version keeps running.	Do not run migrations in CMD or entrypoint. With multiple VMs, all attempt to migrate simultaneously, causing lock conflicts and potential data corruption.
✓ DO	✗ DO NOT
Always configure a [checks] health check block pointing to a lightweight endpoint that returns HTTP 200.	Do not deploy without a health check. Fly cannot determine when a new deployment is ready to receive traffic, risking broken instances receiving user requests.

11.2 Docker

✓ DO	✗ DO NOT
COPY requirements.txt before COPY . . — Docker caches the pip install layer until requirements.txt changes, making code-only deploys significantly faster.	Do not COPY . . before installing Python packages. Every code change busts the pip install cache layer and re-downloads all packages on each deploy.
✓ DO	✗ DO NOT
Create a comprehensive .dockerignore excluding .git, .env*, __pycache__, *.pyc, *.sqlite3.	Do not skip .dockerignore. Without it, .env files containing development secrets may be included in the Docker image pushed to Fly's registry.

11.3 Database & Connections

✓ DO	✗ DO NOT
Use fly postgres attach — it creates a least-privilege database user automatically. The app connects as this user, not as the postgres superuser.	Do not connect to Postgres using the postgres superuser from application code. Use the dedicated user created by fly postgres attach.
✓ DO	✗ DO NOT
Set a statement_timeout at the connection level (30–60 seconds). This prevents runaway queries from blocking your database.	Do not add NOT NULL columns without a default to large tables in a single ALTER TABLE — it locks the entire table. Use a multi-step migration.

11.4 FastAPI Specific

✓ DO	✗ DO NOT
Always pass --proxy-headers and --forwarded-allow-ips "*" to uvicorn in production. Required for correct client IP detection, rate limiting, and geo-IP through Fly's proxy.	Do not use --reload in production uvicorn. It watches for file changes and restarts workers — causing unexpected downtime and wasted CPU on production VMs.
✓ DO	✗ DO NOT
Convert DATABASE_URL from postgresql:// to postgresql+asyncpg:// when using async SQLAlchemy. The Fly-injected URL uses the sync scheme by default.	Do not use synchronous database drivers (psycopg2, asyncpg sync mode) in async FastAPI request handlers. They block the event loop and eliminate async performance benefits.

12. Production Launch Checklist

Complete all items before routing production traffic to a Fly.io deployment.

Build & Docker

- Dockerfile builds locally: docker build -t test . && docker run -p 8000:8000 test
- .dockerignore excludes .git, .env*, __pycache__, *.pyc, *.sqlite3
- App runs as non-root user (RUN useradd -m appuser && USER appuser)
- collectstatic (Django) runs in Dockerfile before EXPOSE

fly.toml

- force_https = true set in [http_service]
- Health check endpoint exists and returns HTTP 200 unconditionally
- [checks] block configured with correct path and port
- min_machines_running = 1 for all user-facing services
- [deploy] release_command set for database migrations
- No secrets in [env] block — only non-sensitive config

Secrets & Configuration

- SECRET_KEY set via fly secrets set (not in fly.toml)
- DATABASE_URL injected via fly postgres attach or fly secrets set
- REDIS_URL set via fly redis attach or fly secrets set
- ALLOWED_HOSTS includes the fly.dev subdomain and all custom domains (Django)
- DEBUG = False confirmed in production settings
- .env* files added to .gitignore

Database

- fly postgres attach run — dedicated user created, DATABASE_URL injected
- Connection pool settings configured (conn_max_age, pool_size, pool_pre_ping)
- Statement timeout set (30–60 seconds)
- Migrations tested on staging environment before production
- No superuser credentials in application connection string

Scaling & Reliability

- Zero-downtime deploy tested: fly deploy while watching fly status
- Health checks pass after deploy: fly checks list
- VM memory usage checked under load: fly metrics
- Custom domain added and TLS certificate issued: fly certs add
- Error monitoring connected (Sentry or equivalent)

13. Conclusion

Fly.io does not attempt to replicate the breadth of AWS or GCP. It makes a different bet: that most application deployment needs can be satisfied by a smaller, more coherent set of primitives — Docker containers, Firecracker VMs, WireGuard networking, and a declarative configuration file — delivered through an interface designed for application developers rather than infrastructure engineers.

For Django DRF and FastAPI applications, Fly.io delivers three concrete advantages over traditional cloud platforms:

1. Time-to-production: A complete deployment with Postgres and Redis can be achieved in under 15 minutes, compared to 2–6 hours for an equivalent AWS ECS setup.
2. Operational simplicity: fly.toml, fly deploy, fly secrets set, and fly postgres attach are the complete operational vocabulary for most API workloads. There is no VPC, IAM, or load balancer configuration required.
3. Cost efficiency: The absence of NAT Gateways, Application Load Balancers, and minimum RDS instance fees reduces the monthly cost floor from $70–120 to $5–15 for a typical production setup.

The platform's tradeoffs are real: Fly Postgres requires more operational involvement than RDS, there is no equivalent to AWS SageMaker or BigQuery for specialized workloads, and enterprise compliance certifications are more limited than AWS. Teams with compliance requirements or dedicated platform engineering should evaluate those factors carefully.

For the majority of Python API workloads — SaaS backends, mobile APIs, internal tooling, and data services operating at startup to mid-scale — Fly.io offers a significantly better default than the complexity overhead of traditional cloud platforms. The configuration patterns and operational practices documented in this whitepaper provide a complete foundation for production deployments.

> **Getting Started**
> Install flyctl, run fly auth login, then fly launch in your project directory. Fly detects Django and FastAPI projects automatically and generates initial configuration. Full configuration templates, Dockerfiles, and code examples are available in the companion GitHub repository.

References & Further Reading

- Fly.io Documentation: https://fly.io/docs/
- Fly.io Postgres Guide: https://fly.io/docs/postgres/
- Django Deployment Checklist: https://docs.djangoproject.com/en/5.0/howto/deployment/checklist/
- FastAPI Deployment: https://fastapi.tiangolo.com/deployment/
- Firecracker VMM: https://firecracker-microvm.github.io/
- WireGuard Protocol: https://www.wireguard.com/
- Patroni (Postgres HA): https://patroni.readthedocs.io/
- Twelve-Factor App: https://12factor.net/