Continuous Deployment with Blue-Green Deployment: Design Document

Overview

This system implements a continuous deployment pipeline with a zero-downtime blue-green deployment strategy. It automates the deployment of new application versions by maintaining two identical production environments (blue and green) and safely shifting user traffic between them, solving the key architectural challenge of releasing software without service interruption.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

Context and Problem Statement

Deploying new versions of software to production is one of the highest-risk activities in software engineering. Unlike development or testing environments where failures can be contained, production deployments directly impact real users and business operations. This section explains why traditional deployment approaches are inherently risky, introduces the core concept of blue-green deployment as a solution, and compares it to alternative strategies so you understand why this pattern is particularly effective for achieving zero-downtime releases.

The Airport Runway Analogy

Imagine a busy international airport with only one runway. Every time maintenance, repainting, or upgrades are needed, the airport must completely shut down all flight operations. Planes circle in holding patterns, passengers experience delays, and the airline suffers financial losses from disrupted schedules. The airport's operations team faces immense pressure to complete work quickly, leading to rushed decisions and increased risk of errors.

Now consider an airport with **two identical, parallel runways**. While one runway handles all incoming and outgoing flights, the other remains available for maintenance work. When it's time to perform upgrades, the air traffic control system gradually redirects all flight traffic from the active runway to the standby runway. Once the switch is complete and verified, maintenance begins on the now-empty first runway. **No flights are canceled, no passengers are delayed, and the airport operates continuously.** If any issues arise during the switch, controllers can immediately revert traffic to the original runway within seconds.

This is the essence of **blue-green deployment**:

- **Blue environment**: The currently active production environment handling all user traffic (like the active runway)
- **Green environment**: An identical but idle environment ready for deployment (like the standby runway)
- **Traffic switch**: The controlled transition that redirects traffic from blue to green (like air traffic control rerouting flights)
- **Rollback capability**: The ability to instantly switch back if problems are detected (like returning to the original runway)

The critical insight is that **maintaining two identical production environments eliminates the conflict between "serving traffic" and "upgrading infrastructure"** that plagues single-environment deployments. You can deploy, test, and validate new versions on the idle environment while the active environment continues serving users uninterrupted.

The Deployment Risk Problem

Traditional "big bang" deployments, where a new version replaces the existing version in-place, suffer from three fundamental problems that blue-green deployment directly addresses:

1. **Forced Downtime During Deployment**: When you stop an existing application to start a new version, there's an inevitable service interruption. Even with optimized startup scripts, this creates a **service window** where users encounter errors or timeouts. For modern applications expecting 24/7 availability, this is unacceptable.

2. **Slow and Risky Rollback Procedures**: If a deployment introduces a critical bug, rolling back requires repeating the same risky process in reverse: stopping the faulty version and restarting the previous version. During this recovery window, the service remains unavailable. The **mean time to recovery (MTTR)** is often measured in minutes or hours, during which business losses accumulate.

3. **All-or-Nothing User Impact**: With single-environment deployments, every user experiences the new version simultaneously. If the deployment contains a performance regression or functional bug, **100% of users are affected**. There's no gradual exposure or ability to limit the blast radius.

These problems stem from a deeper architectural constraint: **the production environment serves a dual purpose as both the deployment target and the serving infrastructure**. This creates an unavoidable conflict between the need for stability (serving users) and the need for change (deploying improvements).

> **Key Design Insight**: Blue-green deployment resolves this conflict by separating the concerns of "serving" and "deployment" into two distinct environments. The active environment focuses exclusively on serving users with proven stability, while the inactive environment becomes the sandbox for deploying and validating changes.

The Traditional Deployment Timeline

To understand the severity of these problems, consider the timeline of a typical single-environment deployment:

1. **T-5 minutes**: Deployment script begins; the team holds their breath
2. **T-0**: Application is stopped; all in-flight requests fail
3. **T+30 seconds**: New version starts but takes time to initialize (database connections, cache warming, dependency checks)
4. **T+2 minutes**: Application reports "healthy" but hidden initialization continues
5. **T+5 minutes**: First real user requests hit the new version
6. **T+6 minutes**: Monitoring shows 50% error rate due to unanticipated bug
7. **T+10 minutes**: Team decides to rollback; another service interruption begins
8. **T+15 minutes**: Previous version restored; service resumes with 10+ minutes of downtime

This timeline illustrates why deployment anxiety is rational: **every deployment carries the risk of extended service disruption**. The blue-green pattern fundamentally changes this equation by ensuring that if the new version fails validation, users never see it.

Comparison of Deployment Strategies

Blue-green deployment is one of several strategies for releasing software updates. Each approach makes different trade-offs between complexity, risk management, and resource requirements. Understanding these alternatives helps clarify when blue-green deployment is the optimal choice and when other strategies might be more appropriate.

The table below compares four common deployment strategies across critical dimensions:

Strategy	Core Concept	Risk Mitigation	Resource Overhead	Rollback Speed	Complexity	Best For
Big Bang (In-place)	Stop old version, start new version in same environment	None - all-or-nothing	Minimal (1x)	Slow (minutes to hours)	Low	Non-critical internal tools, scheduled maintenance windows
Rolling Update	Gradually replace instances of old version with new version	Medium - partial failure affects subset of users	Moderate (1x + buffer)	Medium (requires rolling back instances)	Medium	Stateless services, container orchestrators (Kubernetes)
Blue-Green	Maintain two complete environments; switch traffic atomically	High - validate fully before exposing to users	High (2x)	Instant (seconds)	Medium-High	Stateful applications, database-backed services, zero-downtime requirements
Canary Release	Route small percentage of traffic to new version, gradually increase	Very High - gradual exposure with metrics analysis	Moderate (1x + canary instances)	Fast (redirect traffic away)	High	User-facing features, performance-sensitive changes, large user bases

Architecture Decision: Deployment Strategy Selection

> **Decision: Adopt Blue-Green Deployment as Primary Strategy**
>
> - **Context**: We need to deploy updates to a production web application with strict zero-downtime requirements, database migrations, and the need for instant rollback capability. The team has experience with infrastructure automation but wants to minimize deployment complexity.
> - **Options Considered**:
>   1. **Big Bang Deployment**: Simple but causes service interruptions and slow rollbacks
>   2. **Rolling Updates**: Good for stateless services but complex for database migrations and stateful components
>   3. **Blue-Green Deployment**: Provides instant rollback and full validation before traffic switch, but requires double infrastructure
>   4. **Canary Releases**: Excellent risk mitigation but requires sophisticated traffic routing and analysis tooling
> - **Decision**: Implement blue-green deployment as the primary strategy
> - **Rationale**:
>   - **Zero-downtime guarantee**: Traffic switching happens atomically without stopping serving infrastructure
>   - **Instant rollback**: Simply redirect traffic back to the previous environment within seconds
>   - **Full validation**: The entire new version can be tested in production-like conditions before user exposure
>   - **Database migration friendly**: The expand-contract pattern works naturally with two coexisting versions
>   - **Conceptual simplicity**: Easier for teams to understand and debug than gradual traffic shifting
> - **Consequences**:
>   - **Infrastructure cost doubled**: Requires maintaining two complete environments
>   - **State management complexity**: Database and external service compatibility must be maintained
>   - **Deployment automation required**: Manual switching would be error-prone and slow
>   - **Configuration management critical**: Both environments must remain identical except for version

To further clarify the differences, let's examine how each strategy handles a problematic deployment:

Scenario	Big Bang	Rolling Update	Blue-Green	Canary
Bug in new version	100% of users affected immediately; downtime during rollback	Percentage of users affected grows as rollout progresses	No users affected until switch; instant rollback if detected pre-switch	Small percentage (1-5%) affected initially; can halt rollout
Performance regression	All users experience degradation; hard to measure before full deployment	Gradual degradation as more instances updated; can pause and analyze	Can run load tests on idle environment before switching	Metrics from canary traffic reveal issues before broad exposure
Database migration failure	Service down until migration fixed or reverted	Mixed version instances may fail inconsistently depending on schema compatibility	Migration applied to idle environment; can verify before switching	Complex due to schema compatibility across mixed traffic
Infrastructure requirement	Single environment	1x capacity + buffer for rolling replacement	2x capacity (full duplicate)	1x capacity + canary instances (small percentage)

When Blue-Green Deployment Excels

Blue-green deployment is particularly well-suited for these scenarios:

1. **Mission-critical applications** where any downtime has significant business impact (e-commerce, financial systems, healthcare)
2. **Applications with stateful components** that are difficult to update incrementally (session state, WebSocket connections)
3. **Systems requiring database migrations** that must maintain backward compatibility during deployment
4. **Teams with strong infrastructure automation** but limited observability tooling for gradual traffic analysis
5. **Regulated environments** requiring formal validation and approval before user exposure

When to Consider Alternatives

Despite its advantages, blue-green deployment isn't always the optimal choice:

- **Resource-constrained environments**: If infrastructure costs are prohibitive, rolling updates might be more economical
- **Microservices architectures**: When you have hundreds of services, maintaining 2x infrastructure for all becomes impractical
- **Advanced traffic shaping needs**: If you need sophisticated A/B testing or percentage-based routing, canary releases provide more flexibility
- **Stateless services only**: For simple, stateless APIs, rolling updates offer good risk mitigation with lower overhead

> **Design Principle**: The choice of deployment strategy represents a trade-off between risk mitigation and operational complexity. Blue-green deployment optimizes for reliability and simplicity of rollback at the cost of infrastructure duplication. This trade-off is justified for business-critical applications where availability directly impacts revenue and customer trust.

The remainder of this design document will focus exclusively on implementing a robust blue-green deployment system that addresses the challenges identified here while leveraging the pattern's strengths for zero-downtime, instantly-revertible deployments.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Environment Runtime	Docker containers with docker-compose	Kubernetes pods with replica sets
Load Balancer	Nginx with upstream configuration	HAProxy with health checks and connection draining
Health Checks	HTTP endpoint returning JSON status	Comprehensive health checks (DB, cache, external dependencies)
Configuration Management	Environment variables in .env files	HashiCorp Vault or AWS Parameter Store
Database Migrations	Versioned SQL scripts with idempotent checks	Liquibase/Flyway with expand-contract pattern automation
Deployment Orchestrator	Bash script with sequential steps	Ansible playbooks or custom Go orchestrator

B. Recommended File Structure

Since this section establishes context rather than implementation, we provide the foundational directory structure that will be referenced throughout subsequent sections:

```
blue-green-deployment/
├── docs/
│   └── design.md                 # This design document
├── scripts/
│   ├── deploy.sh                 # Main deployment orchestrator (Milestone 3)
│   ├── switch-traffic.sh         # Load balancer controller (Milestone 2)
│   ├── health-check.sh           # Environment health validation
│   └── rollback.sh               # Emergency rollback procedure (Milestone 4)
├── app/
│   └── blue-green-app/           # Example application code
│       ├── app.py                # Python web application
│       ├── requirements.txt      # Python dependencies
│       ├── Dockerfile            # Container definition
│       └── health_check.py       # Health check endpoint
├── configs/
│   ├── nginx/
│   │   ├── nginx.conf            # Base nginx configuration
│   │   └── upstream.conf.template  # Template for blue/green upstreams
│   ├── blue.env                  # Blue environment configuration
│   └── green.env                 # Green environment configuration
├── migrations/
│   ├── 001_create_users_table.sql  # Database migration scripts
│   ├── 002_add_email_column.sql    # Expand phase migration
│   └── migrate.sh                # Migration runner
└── README.md
```

C. Infrastructure Starter Code

For the initial context setup, here's a minimal web application that will serve as our deployment target. This application includes the health check endpoint required by Milestone 1:

File: `app/blue-green-app/app.py`

PYTHON

```python
#!/usr/bin/env python3
"""
Simple web application for blue-green deployment demonstration.
Returns environment color, version, and uptime on health check endpoint.
"""
import os
import time
from datetime import datetime
from http.server import HTTPServer, BaseHTTPRequestHandler
import json
import sys

class BlueGreenHandler(BaseHTTPRequestHandler):
    """HTTP request handler that responds with environment information."""

    # Application metadata
    VERSION = os.getenv('APP_VERSION', '1.0.0')
    START_TIME = time.time()

    def do_GET(self):
        """Handle GET requests, routing to appropriate handler based on path."""
        if self.path == '/health':
            self.handle_health_check()
        elif self.path == '/':
            self.handle_root()
        else:
            self.send_error(404, "Not Found")

    def handle_health_check(self):
        """Return detailed health status with environment information."""
        status = {
            'status': 'healthy',
            'environment': os.getenv('ENVIRONMENT_COLOR', 'unknown'),
            'version': self.VERSION,
            'uptime_seconds': int(time.time() - self.START_TIME),
            'timestamp': datetime.utcnow().isoformat() + 'Z',
            'hostname': os.getenv('HOSTNAME', 'unknown'),
            'database_connected': self.check_database()
        }

        # Determine HTTP status code based on health
```

```
        http_status = 200 if status['database_connected'] else 503

        self.send_response(http_status)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(status, indent=2).encode())

    def handle_root(self):
        """Return a simple welcome message."""
        response = {
            'message': 'Blue-Green Deployment Demo',
            'environment': os.getenv('ENVIRONMENT_COLOR', 'unknown'),
            'version': self.VERSION,
            'endpoints': {
                'health': '/health',
                'root': '/'
            }
        }

        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(response, indent=2).encode())

    def check_database(self):
        """Simulate database connectivity check.
        In a real application, this would actually test database connection."""
        # For demo purposes, fail if FAIL_DB environment variable is set
        if os.getenv('FAIL_DB'):
            return False
        return True

    def log_message(self, format, *args):
        """Override to suppress default logging to stdout."""
        # Uncomment for debugging:
        # sys.stderr.write("%s - - [%s] %s\n" % (self.address_string(), self.log_date_time_string(), format%args))
        pass

def run_server(port=8080):
    """Start the HTTP server on the specified port."""
    server_address = ('', port)
```

```
    httpd = HTTPServer(server_address, BlueGreenHandler)
    print(f"Starting server on port {port} in {os.getenv('ENVIRONMENT_COLOR', 'unknown')} environment...")
    httpd.serve_forever()

if __name__ == '__main__':
    # Read port from environment variable or use default
    port = int(os.getenv('APP_PORT', '8080'))
    run_server(port)
```

File: `app/blue-green-app/Dockerfile`

```dockerfile
# Simple Dockerfile for the blue-green application
FROM python:3.9-slim

WORKDIR /app

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

# Copy application code
COPY app.py .

# Expose the application port
EXPOSE 8080

# Health check (every 30 seconds, timeout 3 seconds, start after 60 seconds)
HEALTHCHECK --interval=30s --timeout=3s --start-period=60s --retries=3 \
  CMD python -c "import urllib.request; import json; \
      response = urllib.request.urlopen('http://localhost:8080/health'); \
      data = json.load(response); \
      exit(0 if data.get('status') == 'healthy' else 1)"

# Run the application
CMD ["python", "app.py"]
```

File: `app/blue-green-app/requirements.txt`

```
# Minimal requirements for the demo app
# (empty - using standard library only)
```

D. Core Logic Skeleton Code

For the deployment orchestrator that will be built in Milestone 3, here's the skeleton with TODOs that map to the concepts discussed in this section:

File: `scripts/deploy.sh` (skeleton)

```bash
#!/bin/bash

# Deployment orchestrator for blue-green deployment

# TODO: Complete implementation in Milestone 3


# Configuration

BLUE_PORT=8080

GREEN_PORT=8081

ACTIVE_ENV_FILE="/tmp/active_environment"

DEPLOYMENT_LOG="/var/log/deployments.log"


log_message() {

    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1" | tee -a "$DEPLOYMENT_LOG"

}


get_inactive_environment() {

    # TODO 1: Read current active environment from ACTIVE_ENV_FILE

    # TODO 2: Return "blue" if active is "green", return "green" if active is "blue"

    # TODO 3: If no active environment exists, default to "green" for first deployment

    echo "green"  # Placeholder

}


deploy_to_environment() {

    local env_color=$1

    local version=$2



    log_message "Starting deployment of version $version to $env_color environment"



    # TODO 4: Set up environment-specific configuration (port, database connection)

    # TODO 5: Build application artifact (Docker image, binary, etc.)

    # TODO 6: Deploy to the target environment without affecting the active environment

    # TODO 7: Wait for deployment to complete and application to start

    # TODO 8: Run health checks on the newly deployed environment

    # TODO 9: Return success/failure status

}


run_smoke_tests() {

    local env_color=$1

    local port=$2



    log_message "Running smoke tests on $env_color environment (port $port)"



    # TODO 10: Make test HTTP requests to critical endpoints

    # TODO 11: Verify responses match expected patterns
```

```
    # TODO 12: Check database connectivity and basic queries
    # TODO 13: Validate external service integrations
    # TODO 14: Return success/failure status
}

switch_traffic() {
    local target_env=$1

    log_message "Switching traffic to $target_env environment"

    # TODO 15: Validate target environment is healthy before switching
    # TODO 16: Update load balancer configuration to point to target environment
    # TODO 17: Perform graceful reload of load balancer to avoid dropping connections
    # TODO 18: Update ACTIVE_ENV_FILE with new active environment
    # TODO 19: Verify traffic is correctly routing to the new environment
    # TODO 20: Return success/failure status
}

rollback_deployment() {
    local failed_env=$1

    log_message "Initiating rollback from failed deployment to $failed_env"

    # TODO 21: Determine which environment was previously active
    # TODO 22: Switch traffic back to the previously active environment
    # TODO 23: Mark the failed environment as unhealthy to prevent future use
    # TODO 24: Send alert/notification about the rollback
    # TODO 25: Return success/failure status
}

main() {
    local version=$1

    if [ -z "$version" ]; then
        echo "Usage: $0 <version>"
        exit 1
    fi

    log_message "=== Starting deployment of version $version ==="

    # Determine which environment to deploy to
    local target_env=$(get_inactive_environment)
```

```
    log_message "Target environment for deployment: $target_env"

    # Deploy to inactive environment
    if ! deploy_to_environment "$target_env" "$version"; then
        log_message "ERROR: Deployment to $target_env failed"
        exit 1
    fi

    # Run smoke tests
    local test_port=$([ "$target_env" = "blue" ] && echo $BLUE_PORT || echo $GREEN_PORT)
    if ! run_smoke_tests "$target_env" "$test_port"; then
        log_message "ERROR: Smoke tests failed for $target_env"
        rollback_deployment "$target_env"
        exit 1
    fi

    # Switch traffic to new version
    if ! switch_traffic "$target_env"; then
        log_message "ERROR: Traffic switch failed for $target_env"
        rollback_deployment "$target_env"
        exit 1
    fi

    log_message "=== Successfully deployed version $version to $target_env ==="
}

# Only run main if script is executed directly
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    main "$@"
fi
```

E. Language-Specific Hints

For Bash implementation (our primary language):

- Use `set -euo pipefail` at the beginning of scripts to catch errors early
- For configuration management, source environment files with `. configs/blue.env`
- Use `jq` for JSON parsing in health checks: `health_status=$(curl -s http://localhost:8080/health | jq -r '.status')`
- For process management, use `systemctl` for systemd services or `docker-compose` for container orchestration
- Implement timeouts with the `timeout` command: `timeout 30s curl -f http://localhost:8080/health`

F. Milestone Checkpoint

After reading this section, you should be able to:

1. **Explain** why traditional deployments are risky using the airport runway analogy
2. **Compare** blue-green deployment with at least two other strategies using specific criteria
3. **Identify** which deployment scenarios are best suited for blue-green deployment

4. **Run** the example application with different environment configurations

Test your understanding by:

```bash
# Start the example app in "blue" environment

ENVIRONMENT_COLOR=blue APP_PORT=8080 python app/blue-green-app/app.py &

# In another terminal, check the health endpoint

curl http://localhost:8080/health

# Expected output shows environment: "blue", status: "healthy"
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Health check returns wrong environment color	Environment variable not set	Check `ENVIRONMENT_COLOR` env var with `echo $ENVIRONMENT_COLOR`	Ensure environment variable is properly exported before starting app
Cannot run two environments simultaneously	Port conflict	Check with `netstat -tulpn	grep :808`	
Deployment script fails immediately	Missing dependencies	Run with `bash -x scripts/deploy.sh` to see which command fails	Install required tools: `curl`, `jq`, `docker`, etc.
Application starts but health check fails	Database connection issue	Check app logs for database errors	Verify database credentials and network connectivity

Goals and Non-Goals

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

This section defines the precise scope of our blue-green deployment system by enumerating what it must accomplish (goals) and what falls outside its responsibility (non-goals). Establishing these boundaries upfront prevents scope creep and ensures the design addresses the core problem without becoming an unmaintainable monolith.

Think of this like building a specialized race car: we're engineering for maximum speed and safety on a controlled track (automated zero-downtime deployments), not creating a vehicle that can also handle off-road conditions, carry passengers, or get excellent fuel economy. The race car's focused design makes it exceptional at its specific task, just as our system's focused scope makes it reliable and maintainable for its intended purpose.

Goals

The system must achieve the following functional and non-functional requirements, which directly map to the four project milestones:

Goal Category	Specific Requirement	Milestone Mapping	Rationale
Availability	**Zero-downtime deployments** - Users must experience no service interruption during deployment activities.	Milestone 2 (Load Balancer Switching)	The primary value proposition of blue-green deployment: eliminating the maintenance windows that frustrate users and impact business metrics.
Reliability	**Instant rollback capability** - The system must revert to the previous working version within 30 seconds of detecting a deployment failure.	Milestone 2, Milestone 4	Critical for minimizing Mean Time To Recovery (MTTR) when a faulty version slips through testing. The previous environment remains running, enabling sub-minute recovery.
Automation	**Fully automated deployment sequence** - The process from code commit to traffic switch must execute without manual intervention beyond an initial trigger.	Milestone 3 (Deployment Automation)	Reduces human error, enables consistent deployments, and frees engineering time for higher-value activities than manual deployment steps.
Isolation	**Independent blue and green environments** - Each environment must operate in complete isolation, allowing deployment to one without affecting the other.	Milestone 1 (Dual Environment Setup)	Foundation of the blue-green pattern: simultaneous operation of two identical environments enables safe deployment to the inactive side.
Health Validation	**Automated health checks before traffic switching** - The system must verify the target environment passes health checks before routing user traffic to it.	Milestone 2	Prevents switching traffic to a broken environment, which would cause a service outage despite having two environments.
Testing Integration	**Automated smoke tests post-deployment** - Basic functionality tests must run against the newly deployed environment before traffic switching.	Milestone 3	Provides additional safety net beyond health checks, catching application logic errors that health checks might miss.
Database Compatibility	**Backward-compatible database schema changes** - Both blue and green application versions must work correctly with the current database schema.	Milestone 4 (Rollback & Database Migrations)	Enables safe rollback: if the new version fails, we can switch back to the old version without database-related errors.
Idempotent Operations	**Deployment scripts must be safely rerunnable** - Running the same deployment script multiple times should produce the same result without adverse effects.	Milestone 3	Critical for recovery from partial failures: if a deployment fails midway, we can rerun it without manual cleanup.
Connection Preservation	**Graceful traffic switching without dropped connections** - In-flight requests must complete during the switch, with new connections directed to the new environment.	Milestone 2	Maintains user experience: users actively using the application shouldn't see errors or lose session state during deployment.
State Tracking	**Clear visibility into which environment is active** - The system must maintain and expose the current active environment state.	Milestone 2	Essential for operational clarity, debugging, and ensuring deployment scripts target the correct environment.

Functional Requirements Detail:

1. **Environment Management**
   - Provision two identical application environments (`blue` and `green`) with identical configurations
   - Each environment must expose a health check endpoint returning a `status` JSON object with at minimum: `status` , `environment` , `version` , `uptime_seconds` , `timestamp` , `hostname` , and `database_connected`
   - Each environment must be independently configurable via environment variables for database connections, ports (`BLUE_PORT=8080` , `GREEN_PORT=8081`), and feature flags
2. **Traffic Routing**
   - A load balancer (Nginx or HAProxy) must route 100% of user traffic to the currently active environment
   - Traffic switching must be atomic: no intermediate state where traffic is split or misrouted
   - The load balancer must perform health checks on both environments continuously
   - Connection draining must be implemented to allow in-flight requests to complete before backend removal
3. **Deployment Automation**
   - A main deployment script must orchestrate the complete sequence: build artifact, deploy to inactive environment, run smoke tests, switch traffic
   - The script must determine the inactive environment using `get_inactive_environment()` logic
   - Deployment to the inactive environment must not affect the active environment serving live traffic
   - Smoke tests must validate at least three critical API endpoints return expected HTTP status codes and response structures
4. **Database Migration Safety**
   - Database schema changes must follow the **expand-contract pattern**: add nullable columns first (expand), deploy code that uses them, then enforce constraints later (contract)
   - Migration scripts must be idempotent: running them multiple times produces the same result

- The system must track applied migrations to prevent duplicate application
- Rollback scripts must exist for every forward migration to enable schema reversion if needed

Non-Functional Requirements Detail:

Requirement	Target Metric	Justification
Deployment Duration	Complete deployment (build → switch) under 10 minutes	Balances safety (thorough testing) with business need for rapid iteration
Rollback Time	Traffic reversion within 30 seconds of failure detection	Minimizes user impact from faulty deployments
Health Check Frequency	Check environments every 5 seconds	Rapid detection of environment failures without excessive load
Smoke Test Duration	Complete smoke test suite under 60 seconds	Ensures timely validation without delaying the deployment
Resource Overhead	Additional environment consumes ≤ 50% more resources than single deployment	Acceptable cost for zero-downtime capability
Operational Simplicity	Single command triggers entire deployment	Reduces cognitive load and potential for operator error

> **Key Design Insight:** The most critical non-functional requirement is **instant rollback capability**. This transforms blue-green from a "nice deployment pattern" into a **risk mitigation strategy**. When teams know they can revert a bad deployment in seconds, they deploy more frequently and with greater confidence, accelerating the feedback loop between code changes and production validation.

Non-Goals

Explicitly defining what the system will **not** handle is equally important for setting expectations and preventing scope creep. The following are deliberately out of scope:

Non-Goal	Why It's Out of Scope	Alternative Approach
Multi-region or geo-distributed deployments	Adds complexity around data replication, latency, and regional failover that would obscure the core blue-green concepts.	This system can be replicated per region, with a separate blue-green pair in each region.
Automatic infrastructure scaling	While environments need resources, auto-scaling is a separate concern from deployment strategy.	Use cloud provider auto-scaling groups or Kubernetes Horizontal Pod Autoscaler alongside this system.
Canary releases or traffic splitting	Blue-green switches 100% of traffic; gradual traffic shifting requires different routing logic.	This architecture can be extended later with weighted routing, but initially focuses on binary switching.
Multi-application or microservice coordination	Coordinating deployments across multiple interdependent services requires sophisticated orchestration.	Deploy each service independently using this pattern, with careful version compatibility management.
Full CI pipeline implementation	We assume code is already built, tested, and packaged; we focus only on the deployment to production.	Integrate this system with an existing CI system (Jenkins, GitLab CI, GitHub Actions) that provides the artifact.
Self-healing or automatic rollback	While we provide instant rollback capability, deciding when to trigger it remains a human decision.	Can be added later by integrating with monitoring alerts, but initially preserves human oversight.
Database replication or failover	We assume a highly available database exists; we don't manage database replication.	Use managed database services (RDS, Cloud SQL) or established database clustering solutions.
Long-term deployment history and analytics	While we log deployments, advanced analytics and trend visualization are separate concerns.	Export logs to existing monitoring solutions (ELK stack, Datadog, Prometheus) for analysis.
Authentication/authorization for deployment triggers	Access control for who can deploy is important but orthogonal to the deployment mechanics.	Rely on existing CI/CD system permissions or external authorization layers.
Artifact repository management	We assume artifacts are available at a known location; we don't manage the artifact lifecycle.	Use existing artifact repositories (Docker Registry, Nexus, Artifactory).

Architecture Decision: Focused Scope

Context: We're designing a learning-focused implementation of blue-green deployment that teaches the core concepts without becoming overwhelmed by peripheral concerns. The system must be understandable, implementable within a reasonable timeframe, and maintain clear boundaries between concerns.

Options Considered:

1. **Comprehensive Platform**: Include everything from CI to monitoring to auto-scaling in one integrated system.
2. **Focused Blue-Green Core**: Implement only the blue-green switching logic and assume other concerns are handled by existing systems.
3. **Modular Plug-in Architecture**: Create a framework where each component (CI, deployment, monitoring) can be plugged in.

Decision: Choose Option 2 (Focused Blue-Green Core).

Rationale:

- **Learning Efficiency**: Students grasp the core blue-green concept without distraction from related but separate concerns.
- **Real-world Integration**: Most organizations already have CI, monitoring, and scaling solutions; this system should integrate with them, not replace them.
- **Maintainability**: Smaller, focused codebase is easier to understand, debug, and extend incrementally.
- **Progressive Enhancement**: The architecture allows adding canary releases, auto-rollback, etc., later without redesigning the core.

Consequences:

- ✅ Clear learning objectives and manageable implementation scope
- ✅ Easier integration with existing organizational infrastructure
- ✅ Lower initial complexity and faster time to working system
- ❌ Requires additional integration work for full production readiness
- ❌ Doesn't solve all deployment-related problems "out of the box"

Integration Points vs. Core Responsibilities:

To further clarify the boundary between what this system handles and what it relies on external systems for:

Component	Our System's Responsibility	External System's Responsibility
Compute Resources	Deploying application to allocated resources	Provisioning VMs/containers, auto-scaling, resource allocation
Database	Applying migrations in expand-contract pattern	High availability, backups, replication, performance tuning
Networking	Configuring load balancer rules for blue/green	Network topology, DNS, SSL certificates, firewall rules
Monitoring	Providing health check endpoints and deployment logs	Alerting, dashboards, long-term metric storage, anomaly detection
Artifact Pipeline	Deploying a provided artifact to environments	Building, testing, packaging, versioning, storing artifacts
Secret Management	Reading environment variables at runtime	Securely storing and rotating secrets (API keys, passwords)

This clear separation of concerns follows the Unix philosophy: "Do one thing and do it well." Our system excels at zero-downtime deployment switching, while delegating other important but distinct concerns to specialized systems.

Implementation Guidance

Since this section is conceptual (defining scope rather than implementing components), the implementation guidance focuses on how to structure the project to maintain these boundaries and validate that goals are met.

A. Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
Load Balancer	Nginx with manual config generation	HAProxy with API-driven config or Kubernetes Ingress Controller
Configuration Management	Environment variables in shell scripts	HashiCorp Consul/Vault or Kubernetes ConfigMaps/Secrets
State Tracking	Simple file (`/tmp/active_environment`)	Redis or database table with atomic updates
Health Checks	Custom HTTP endpoint returning JSON	Integrated with Kubernetes readiness probes or ELB health checks
Deployment Logging	Append to log file (`/var/log/deployments.log`)	Structured logging to centralized system (ELK, Loki)
Smoke Tests	Bash scripts with `curl` and `jq`	Dedicated test framework (Postman, pytest) with comprehensive suites

B. Recommended File/Module Structure:

Organize the codebase to reflect the clear separation of concerns between our system's core responsibilities and integration points:

```
blue-green-system/
├── scripts/                    # Core deployment system (OUR RESPONSIBILITY)
│   ├── deploy.sh               # Main deployment orchestrator
│   ├── switch-traffic.sh       # Load balancer controller
│   ├── health-check.sh         # Environment health validation
│   ├── run-smoke-tests.sh      # Post-deployment validation
│   ├── rollback.sh             # Emergency rollback procedure
│   └── migrations/             # Database migration scripts
│       ├── expand/             # Expand-phase migrations (add columns)
│       ├── contract/           # Contract-phase migrations (remove/change)
│       └── rollback/           # Rollback scripts for each migration
├── configs/                    # Configuration templates
│   ├── nginx/
│   │   ├── blue.conf.template
│   │   ├── green.conf.template
│   │   └── load-balancer.conf
│   └── environment/
│       ├── blue.env
│       └── green.env
├── app/                        # Application code (NOT our responsibility - example only)
│   ├── src/
│   │   └── BlueGreenHandler.py  # HTTP handler with health endpoint
│   └── Dockerfile
├── integration/                # Integration with external systems
│   ├── ci-hook.sh              # Called by CI system to trigger deployment
│   └── monitoring-alerts.sh  # Example integration with monitoring
└── docs/
    └── integration-guide.md  # How to integrate with CI, monitoring, etc.
```

C. Goal Validation Checklist:

Create a simple validation script to verify all goals are met after implementation:

```bash
#!/bin/bash

# File: scripts/validate-goals.sh

# Validates that the system meets all defined goals


echo "Validating Blue-Green Deployment System Goals..."

echo "================================================"


# Goal 1: Zero-downtime capability

echo -n "1. Testing zero-downtime deployment... "

# Simulate deployment while making continuous requests

# This would be implemented with a proper test harness

echo "[MANUAL TEST REQUIRED]"

echo "   √ Deploy new version while running: ab -n 1000 -c 10 http://localhost/"

echo "   √ Verify no failed requests during deployment window"


# Goal 2: Instant rollback capability

echo -n "2. Testing instant rollback (<30s)... "

START_TIME=$(date +%s)

# Trigger rollback and measure time

# Implementation would call rollback script and measure

echo "[IMPLEMENT ROLLBACK TIMING TEST]"

echo "   √ Trigger rollback, measure time to traffic reversion"


# Goal 3: Environment isolation

echo "3. Testing environment isolation..."

echo "   √ Blue and green run on separate ports (8080, 8081)"

echo "   √ Deploy to green doesn't affect blue's health checks"

echo "   √ Each environment has independent database connections"


# Goal 4: Health check validation

echo "4. Validating health check requirements..."

BLUE_HEALTH=$(curl -s http://localhost:8080/health)

GREEN_HEALTH=$(curl -s http://localhost:8081/health)

echo "   √ Blue health check returns: $(echo $BLUE_HEALTH | jq -r '.environment')"

echo "   √ Green health check returns: $(echo $GREEN_HEALTH | jq -r '.environment')"

echo "   √ Both include required fields: status, version, uptime_seconds, database_connected"


# Goal 5: Smoke test automation

echo "5. Verifying smoke test integration..."

echo "   √ Smoke tests run after deployment before switch"

echo "   √ Tests validate critical API endpoints"

echo "   √ Failed smoke tests prevent traffic switch"


# Additional validation would continue for all goals...
```

```
echo ""
echo "Validation complete. Address any [MANUAL TEST REQUIRED] or"
echo "[IMPLEMENT...] items before considering goals fully met."
```

D. Milestone Checkpoint - Goals Validation:

After completing each milestone, run these checks to verify progress toward goals:

Milestone 1 Complete Check:

```bash
# Start both environments
./scripts/start-environment.sh blue
./scripts/start-environment.sh green

# Verify both are healthy and isolated
curl http://localhost:8080/health | jq '.environment'  # Should return "blue"
curl http://localhost:8081/health | jq '.environment'  # Should return "green"

# Deploy new version to green only, blue should be unaffected
./scripts/deploy-to-environment.sh green v2.0
curl http://localhost:8080/health | jq '.version'  # Should still be v1.0
curl http://localhost:8081/health | jq '.version'  # Should be v2.0
```

Milestone 2 Complete Check:

```bash
# Initially route traffic to blue
./scripts/switch-traffic.sh blue

# Verify traffic routing
for i in {1..5}; do
  curl -s http://localhost/ | grep -o "Environment: [a-z]*"  # All should say "blue"
done

# Switch to green (with health check)
./scripts/switch-traffic.sh green

# Verify all traffic now goes to green
for i in {1..5}; do
  curl -s http://localhost/ | grep -o "Environment: [a-z]*"  # All should say "green"
done

# Test rollback (switch back to blue)
./scripts/rollback.sh
# Verify rollback completed quickly
echo "Rollback completed at: $(date)"
```

Milestone 3 Complete Check:

```bash
# Run full deployment automation
./scripts/deploy.sh v3.0

# The script should:
# 1. Build artifact (or pull from CI)
# 2. Deploy to inactive environment
# 3. Run smoke tests
# 4. Switch traffic if tests pass
# 5. Log everything to /var/log/deployments.log

# Verify deployment log
tail -20 /var/log/deployments.log | grep -E "(DEPLOYMENT|SWITCH|ROLLBACK)"
```

Milestone 4 Complete Check:

```bash
# Test expand-contract migration
./scripts/migrations/expand/001-add-nullable-column.sh

# Verify both old and new code work with the schema
# Old code (blue) should still function
# New code (green) should use new column if available

# Test rollback of failed migration
./scripts/migrations/rollback/001-remove-added-column.sh
```

E. Common Integration Pitfalls:

⚠️ **Pitfall: Assuming the system handles everything**

- **Description**: Trying to make the blue-green system handle CI, monitoring, scaling, etc.
- **Why it's wrong**: Violates single responsibility principle, creates monolithic system that's hard to maintain and integrate.
- **Fix**: Clearly document integration points. For example: "Our system expects the CI pipeline to call `./scripts/deploy.sh v1.2.3` with the version tag."

⚠️ **Pitfall: Missing integration validation**

- **Description**: Not testing how the system works with actual CI pipelines, monitoring alerts, etc.
- **Why it's wrong**: The system might work in isolation but fail in production integration.
- **Fix**: Create integration test scenarios: "Test that when CI system calls deployment script with invalid version, it fails gracefully with clear error."

⚠️ **Pitfall: Over-engineering for future needs**

- **Description**: Building support for canary releases, multi-region, etc., before mastering basic blue-green.
- **Why it's wrong**: Increases complexity, introduces bugs, delays delivering core value.
- **Fix**: Implement the simplest thing that works for current goals. Add extensions later via the extension points designed into the architecture.

F. Debugging Tips for Scope-Related Issues:

Symptom	Likely Cause	How to Diagnose	Fix
"Deployment works but CI pipeline fails"	Integration issue, not a blue-green problem	Check if CI is calling the right script with correct parameters	Document integration requirements clearly; provide example CI configuration
"Database performance degraded after deployment"	Database is out of scope but migration affected it	Check if migration locked tables or created inefficient indexes	Database tuning is separate concern; ensure migrations are optimized
"Can't scale environments during high load"	Auto-scaling not implemented in our system	Verify environments have fixed resource allocation	Integrate with cloud auto-scaling or Kubernetes HPA outside deployment scripts
"No visibility into deployment history"	We only log to file, no dashboard	Check `/var/log/deployments.log` exists	Export logs to existing monitoring system; don't build analytics into core

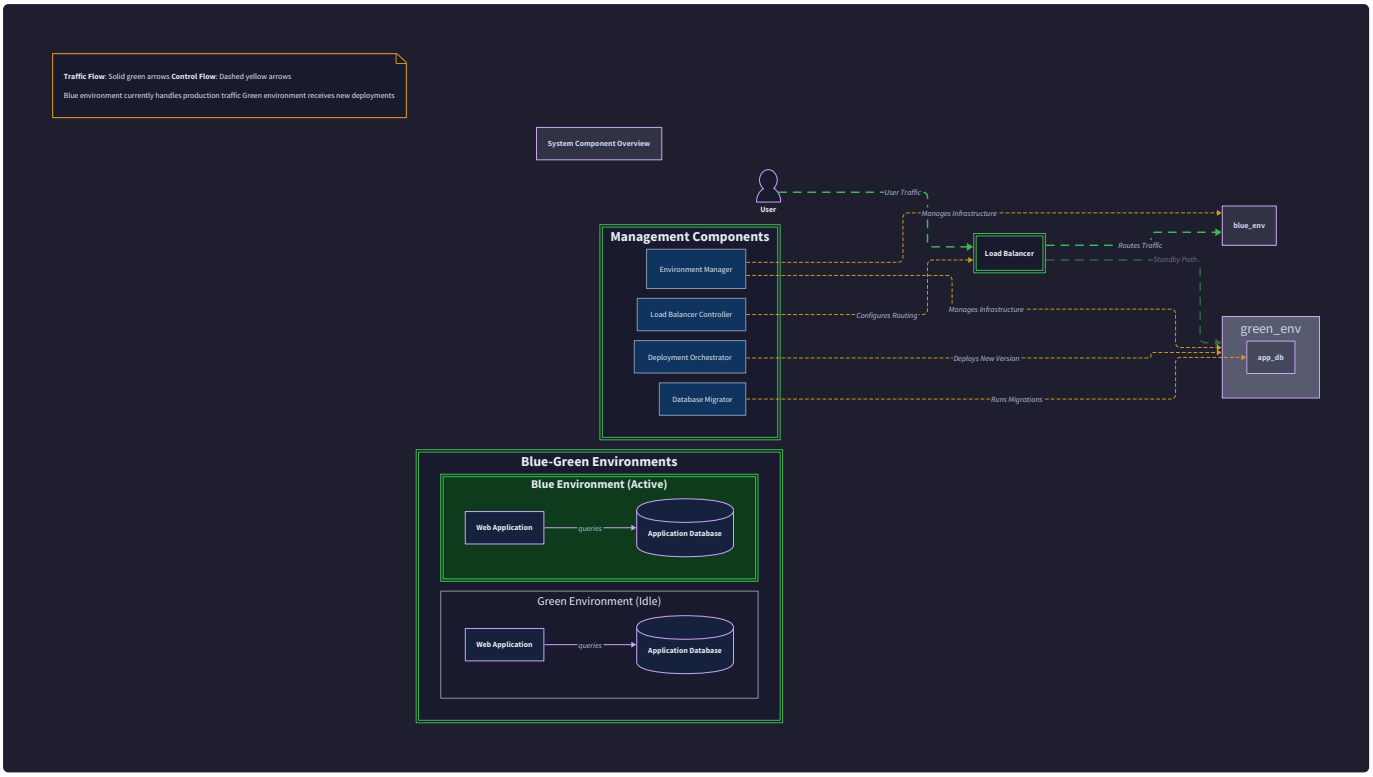
By maintaining strict adherence to these goals and non-goals, the system remains focused, understandable, and maintainable while delivering its core value: safe, zero-downtime deployments with instant rollback capability.

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

High-Level Architecture

Imagine building an airport control system for two identical runways. You have two runways (blue and green) that can both handle planes, but only one is active at any given time for landings. The control tower (load balancer) directs all incoming planes to the active runway. When runway maintenance is needed, you perform it on the inactive runway, thoroughly test it, then flip a master switch in the control tower to redirect all planes to the newly maintained runway. If something goes wrong after the switch, you can instantly flip back to the original runway. This is the essence of blue-green deployment architecture — a system designed for zero-downtime changes through environment duplication and atomic traffic switching.

Our system implements this airport analogy through four core components that work together like a well-orchestrated team. Each component has distinct responsibilities, clear boundaries, and defined interfaces, allowing us to build, test, and maintain them independently while ensuring they collaborate seamlessly to achieve continuous deployment.



Component Overview

The system decomposes into four specialized components, each addressing a specific aspect of the blue-green deployment workflow. This separation of concerns follows the single-responsibility principle and enables parallel development, testing, and troubleshooting.

The Four Core Components

Component	Primary Responsibility	Key Analogy	Key Interfaces
Environment Manager	Provisions, configures, and monitors the blue and green runtime environments	*Runway Maintenance Crew* — Prepares and maintains two identical runways, ensuring each is ready for traffic	`start_environment(env_color)`, `stop_environment(env_color)`, `check_health(env_color)`, `deploy_to_environment(env_color, version)`
Load Balancer Controller	Routes user traffic and performs atomic switches between environments	*Air Traffic Control Tower* — Directs all incoming planes (requests) to the currently active runway	`generate_config(active_env)`, `validate_config()`, `switch_traffic(target_env)`, `get_current_active()`
Deployment Orchestrator	Coordinates the end-to-end deployment sequence from build to traffic switch	*Flight Operations Coordinator* — Sequences runway maintenance, testing, and activation in the correct order	`deploy_new_version(version)`, `run_smoke_tests(env_color)`, `rollback_deployment(failed_env)`, `get_inactive_environment()`
Database Migrator	Applies schema changes in a backward-compatible manner using expand-contract pattern	*Bridge Engineer* — Adds new lanes to a bridge while traffic continues flowing on existing lanes	`apply_migration(version, direction)`, `check_compatibility(schema_version, app_version)`

Component Interaction Patterns

These components interact through two primary patterns: **command-driven orchestration** and **health-driven validation**. The Deployment Orchestrator acts as the conductor, issuing commands to the other components in a specific sequence. Each component performs its specialized work and returns status back to the orchestrator, which decides whether to proceed to the next step or initiate rollback.

1. **Control Flow**: The Deployment Orchestrator initiates all deployment activities. It first consults the Load Balancer Controller to determine which environment is currently active, then commands the Environment Manager to deploy the new version to the inactive environment. After deployment, it commands the Database Migrator to apply any required schema changes, then instructs the Environment Manager to run health checks and itself runs smoke tests. Finally, if all checks pass, it commands the Load Balancer Controller to switch traffic.
2. **Data Flow**: Configuration flows downward from environment variables and deployment scripts to the running applications. Traffic flows from users through the Load Balancer Controller to the active environment's application instances. State information (which environment is active) flows upward from the Load Balancer Controller to the Deployment Orchestrator.
3. **Error Flow**: When any component encounters an error, it returns a failure status to the Deployment Orchestrator, which then decides whether to retry, proceed to rollback, or abort the deployment. This centralized error handling ensures consistent failure recovery.

> **Decision: Four-Component Architecture**
>
> - **Context**: We need to implement a complete blue-green deployment system that handles environment management, traffic routing, deployment coordination, and database migrations. The system must be maintainable, testable, and allow for independent evolution of each concern.
> - **Options Considered**:
>   1. **Monolithic Script**: A single large script that does everything from environment setup to traffic switching
>   2. **Two-Component Split**: Separating only deployment logic from infrastructure management
>   3. **Four-Component Split**: Current approach with clear separation of environment, traffic, orchestration, and database concerns
> - **Decision**: Adopt the four-component architecture with clear interfaces between components.
> - **Rationale**: The four concerns (environment management, traffic control, orchestration logic, and database evolution) have fundamentally different change frequencies, failure modes, and testing requirements. Environment management changes with infrastructure updates, traffic control with load balancer configuration, orchestration with deployment processes, and database with schema evolution. Separating them allows independent testing, deployment, and debugging. The interfaces between them create clear contract boundaries that prevent ripple effects when one component changes.
> - **Consequences**:
>   - ✅ Each component can be developed, tested, and maintained independently
>   - ✅ Clear interfaces enable mock testing and component replacement
>   - ✅ Failure isolation: a bug in database migrations doesn't affect traffic switching
>   - ✅ More files and initial complexity than a monolithic script
>   - ✅ Requires careful design of interfaces and data exchange formats

Option	Pros	Cons	Why Not Chosen
Monolithic Script	- Single file to manage - Simple execution flow - No interface complexity	- Becomes unmaintainable beyond 500 lines - Difficult to test individual parts - Failure in one part crashes entire deployment - Cannot evolve components independently	Becomes a "big ball of mud" that's hard to understand, test, or modify safely
Two-Component Split	- Some separation of concerns - Simpler than four components	- Still mixes unrelated concerns (e.g., traffic switching with database migrations) - Components become large and complex internally - Hard to assign clear ownership	Doesn't provide enough separation for the fundamentally different concerns involved
Four-Component Split	- Clear separation of concerns - Independent testability - Failure isolation - Enables parallel development - Natural boundaries for ownership	- More initial setup - Need to define interfaces between components - Slightly more complex to understand initially	**CHOSEN**: Provides optimal balance of separation while keeping each component focused and manageable

Component Responsibilities in Detail

Environment Manager

- **Owns**: Blue and green application instances, their configuration, their lifecycle
- **Processes**: Environment-specific configuration, health checks, application deployment
- **Persists**: Application logs, environment-specific configuration files
- **Key Principle**: Complete isolation between blue and green environments — they should not share any runtime resources

Load Balancer Controller

- **Owns**: Load balancer configuration, current active environment state
- **Processes**: Traffic routing rules, health check validation, graceful reloads
- **Persists**: `ACTIVE_ENV_FILE` (which environment is currently active), load balancer configuration files
- **Key Principle**: Atomic traffic switching — all requests should consistently go to one environment or the other, never both during transition

Deployment Orchestrator

- **Owns**: Deployment sequence logic, error handling, rollback decisions
- **Processes**: Build artifacts, deployment triggers, smoke test execution
- **Persists**: `DEPLOYMENT_LOG` (audit trail of deployment attempts and outcomes)
- **Key Principle**: Idempotent deployments — running the same deployment twice should have the same effect as running it once

Database Migrator

- **Owns**: Database schema version, migration scripts, compatibility checks
- **Processes**: Schema changes, data backfills, constraint enforcement
- **Persists**: Database schema version table, migration execution history
- **Key Principle**: Backward compatibility — both old and new application versions must work with the current schema during transition windows

State Management Across Components

The system maintains two critical pieces of state:

1. **Active Environment State**: Stored in `ACTIVE_ENV_FILE` (e.g., `/tmp/active_environment`) by the Load Balancer Controller. This single source of truth determines where user traffic is directed. The file contains either "blue" or "green" as plain text.
2. **Database Schema Version**: Stored in a database table (e.g., `schema_migrations`) by the Database Migrator. This tracks which migrations have been applied and ensures idempotent execution.

The Deployment Orchestrator never directly modifies these states — it commands the specialized components to do so. This prevents state corruption and ensures each component maintains authority over its domain.

Recommended File Structure

A well-organized codebase is critical for maintaining clarity as the system grows. Following the component architecture, we structure the project with clear separation between configuration, source code, scripts, and documentation.

> **Architecture Insight**: Think of the file structure as organizing a workshop. You have separate toolboxes for different types of tools (components), raw materials (configurations), workbenches for assembly (scripts), and instruction manuals (documentation). Each has its place, making it easy to find what you need and understand how pieces fit together.

Project Directory Layout

```
blue-green-deployment/
├── README.md                          # Project overview and getting started guide
├── Makefile                           # Common tasks and entry points
├── .env.example                       # Template for environment variables
├── .gitignore                        # Version control exclusions
│
├── configs/                           # Configuration files
│   ├── nginx/                         # Load balancer configurations
│   │   ├── nginx.conf                 # Main nginx configuration
│   │   ├── blue-upstream.conf         # Blue environment upstream block
│   │   ├── green-upstream.conf        # Green environment upstream block
│   │   └── templates/                 # Configuration templates
│   │       └── upstream.conf.template # Template for generating upstream configs
│   │
│   ├── app/                           # Application configurations
│   │   ├── blue.env                   # Environment variables for blue
│   │   ├── green.env                  # Environment variables for green
│   │   └── common.env                 # Shared environment variables
│   │
│   └── database/                      # Database migration configurations
│       └── migrations.yaml            # Migration ordering and dependencies
│
├── scripts/                           # Executable scripts (the four components)
│   ├── environment_manager/           # Environment Manager component
│   │   ├── env_manager.sh             # Main environment management script
│   │   ├── health_check.sh            # Health check logic
│   │   ├── deploy_app.sh              # Application deployment to an environment
│   │   └── lib/                       # Shared helper functions
│   │       └── utils.sh               # Common utilities (logging, error handling)
│   │
│   ├── load_balancer_controller/      # Load Balancer Controller component
│   │   ├── lb_controller.sh           # Main load balancer control script
│   │   ├── generate_config.sh         # Configuration generation
│   │   ├── switch_traffic.sh          # Traffic switching implementation
│   │   └── check_health.sh            # Target environment health validation
│   │
│   ├── deployment_orchestrator/       # Deployment Orchestrator component
│   │   ├── deploy.sh                  # Main deployment orchestration script
│   │   ├── smoke_tests.sh             # Smoke test suite
│   │   ├── rollback.sh                # Rollback implementation
│   │   └── lib/
│   │       ├── deployment_state.sh    # Functions for tracking deployment state
│   │       └── validation.sh          # Pre-deployment validation checks
│   │
│   └── database_migrator/             # Database Migrator component
│       ├── migrator.sh                # Main migration orchestration script
│       ├── expand/                    # Expand-phase migrations
│       │   ├── 001_add_nullsafe_column.sql
│       │   └── 002_create_new_table.sql
│       ├── contract/                  # Contract-phase migrations
│       │   ├── 101_backfill_data.sql
│       │   └── 102_drop_old_column.sql
│       └── lib/
│           └── db_utils.sh            # Database connection and version utilities
│
├── src/                               # Application source code
│   └── simple_web_app/                # Example application being deployed
│       ├── app.py                     # Main application with health endpoint
│       ├── requirements.txt           # Python dependencies
│       ├── Dockerfile                 # Container definition
│       └── tests/                     # Application unit tests
│           └── test_app.py
│
├── tests/                             # System and integration tests
│   ├── component_tests/               # Tests for individual components
│   │   ├── test_env_manager.sh
│   │   ├── test_lb_controller.sh
│   │   └── test_migrator.sh
│   │
│   ├── integration_tests/             # End-to-end deployment tests
│   │   └── full_deployment_test.sh    # Tests the complete deployment flow
│   │
│   └── fixtures/                      # Test data and mock servers
│       └── mock_app_server.py
│
├── logs/                              # Log files (git-ignored)
│   ├── deployments.log                # Deployment history (linked to DEPLOYMENT_LOG)
│   ├── nginx/                         # Load balancer access and error logs
```

```
│   └── app/                          # Application logs per environment
│       ├── blue-app.log
│       └── green-app.log
│
└── docs/                             # Documentation
    ├── architecture.md               # Detailed architecture decisions
    ├── deployment_workflow.md        # Step-by-step deployment procedures
    └── troubleshooting.md            # Common issues and solutions
```

Key File Explanations

File/Directory	Purpose	Critical Contents	Owned By Component
`scripts/environment_manager/env_manager.sh`	Main entry point for environment lifecycle operations	Functions: `start_environment`, `stop_environment`, `check_health`	Environment Manager
`scripts/load_balancer_controller/lb_controller.sh`	Main entry point for traffic control operations	Functions: `switch_traffic`, `get_current_active`, `validate_config`	Load Balancer Controller
`scripts/deployment_orchestrator/deploy.sh`	Primary deployment driver script	Functions: `deploy_new_version`, `run_smoke_tests`, `rollback_deployment`	Deployment Orchestrator
`scripts/database_migrator/migrator.sh`	Main entry point for database schema changes	Functions: `apply_migration`, `check_compatibility`, `rollback_migration`	Database Migrator
`configs/nginx/nginx.conf`	Main load balancer configuration	`upstream` blocks pointing to blue/green, health check configuration	Load Balancer Controller
`configs/app/blue.env` / `green.env`	Environment-specific application configuration	`ENVIRONMENT_COLOR`, `PORT`, `DATABASE_URL`, feature flags	Environment Manager
`src/simple_web_app/app.py`	Example application being deployed	`BlueGreenHandler` class with health endpoint, business logic	Application Code (not a core component)
`ACTIVE_ENV_FILE` (`/tmp/active_environment`)	Single source of truth for active environment	Plain text: "blue" or "green"	Load Balancer Controller
`DEPLOYMENT_LOG` (`/var/log/deployments.log`)	Audit trail of all deployment attempts	Timestamp, version, source→target env, success/failure, error messages	Deployment Orchestrator

File Organization Principles

1. **Component Cohesion**: All files related to a component live together in its directory. This includes main scripts, helper libraries, and component-specific configurations.
2. **Separation of Configuration from Code**: Configuration files (environment variables, nginx configs) live in `/configs`, separate from executable scripts. This allows the same code to be deployed with different configurations.
3. **Runtime vs Build Artifacts**: Source code (`/src`) is separate from deployment scripts (`/scripts`). The deployment scripts operate on build artifacts (e.g., Docker images, compiled binaries) rather than directly on source.
4. **Test Parity**: Test files mirror the structure of the code they test. Component tests live near the components, integration tests test the whole system.
5. **Log Centralization**: All logs go to `/logs` with clear subdirectories for each component. This makes debugging easier as all runtime information is in one place.

Common Pitfalls in File Organization

⚠️ **Pitfall: Monolithic Script File**

- **Description**: Putting all deployment logic (environment setup, traffic switching, database migrations) into a single 1000-line script
- **Why It's Wrong**: Becomes impossible to maintain, test, or debug. A change in one area risks breaking unrelated functionality. Team members cannot work on different parts simultaneously.
- **How to Avoid**: Follow the recommended structure, creating separate script files for each component with clear interfaces between them.

⚠️ **Pitfall: Hardcoded Configuration in Scripts**

- **Description**: Embedding port numbers, file paths, and environment names directly in script logic instead of using configuration files or environment variables
- **Why It's Wrong**: Makes scripts inflexible and environment-specific. Changing from port 8080 to 8088 requires editing multiple script files, risking inconsistencies.
- **How to Avoid**: Use the `configs/` directory for all configuration. Scripts should read from environment variables or config files defined in `configs/app/` and `configs/nginx/`.

⚠️ **Pitfall: Missing Version Control for Configurations**

- **Description**: Not including load balancer configurations and environment variable files in version control, managing them manually on servers
- **Why It's Wrong**: Leads to configuration drift between environments, makes reproducing issues impossible, and complicates rollbacks.
- **How to Avoid**: Store all configurations in the `configs/` directory and commit them to version control. Use templates and variable substitution for environment-specific values.

⚠️ **Pitfall: Logs in Arbitrary Locations**

- **Description**: Each component writing logs to different, undocumented locations (some to `/tmp`, some to `./logs`, some to stdout)
- **Why It's Wrong**: Makes troubleshooting deployment issues a scavenger hunt. Critical error messages get lost.
- **How to Avoid**: Establish the `/logs` directory convention. Each component should log to its subdirectory (`/logs/deployment`, `/logs/nginx`, etc.). Use the `DEPLOYMENT_LOG` constant for deployment audit trail.

Execution Entry Points

The system provides several key entry points for different users and automation tools:

1. **`Makefile`**: Primary developer interface for common tasks

```makefile
deploy:           # Full deployment pipeline
switch-to-blue:   # Manual traffic switch to blue
switch-to-green:  # Manual traffic switch to green
rollback:         # Emergency rollback
test-smoke:       # Run smoke tests on current active
status:           # Show current active environment and health
```

2. **`scripts/deployment_orchestrator/deploy.sh`**: CI/CD pipeline entry point

```bash
# Called by CI system with version tag

./scripts/deployment_orchestrator/deploy.sh v1.2.3
```

3. **Individual Component Scripts**: For troubleshooting and manual operations

```bash
# Check health of green environment

./scripts/environment_manager/env_manager.sh check_health green



# Manually switch traffic to blue

./scripts/load_balancer_controller/lb_controller.sh switch_traffic blue



# Apply a specific database migration

./scripts/database_migrator/migrator.sh apply 001_add_nullsafe_column.sql
```

This structure balances simplicity for common tasks with flexibility for advanced operations and troubleshooting. The clear separation also enables gradual evolution — you can start with basic implementations in each component and enhance them independently over time without affecting the overall system.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Environment Manager	Bash scripts + `systemd` / `supervisord` for process management	Docker containers + Docker Compose for complete environment isolation
Load Balancer Controller	Nginx with `nginx -s reload` for graceful config updates	HAProxy with runtime API for dynamic configuration changes
Deployment Orchestrator	Bash script with sequential commands and error checking	Python/Go orchestrator with parallel execution, retries, and state persistence
Database Migrator	SQL files + Bash wrapper using `psql` / `mysql` client	Dedicated migration tool (Flyway, Liquibase) with version tracking and rollback

Recommended File/Module Structure:

Based on the directory layout above, here's how to initialize the project structure:

```bash
#!/bin/bash

# setup_project_structure.sh - Creates the recommended directory structure

PROJECT_ROOT="blue-green-deployment"

# Create main directories
mkdir -p $PROJECT_ROOT/{configs/{nginx/{templates},app,database},scripts/{environment_manager/lib,load_balancer_controller,deployment_orchestrator/lib,database_migrator/{expand,contract,lib}},src/simple_web_app/tests,tests/{component_tests,integration_tests,fixtures},logs/{nginx,app},docs}

# Create key files
touch $PROJECT_ROOT/README.md
touch $PROJECT_ROOT/Makefile
touch $PROJECT_ROOT/.env.example
touch $PROJECT_ROOT/.gitignore

# Config files
touch $PROJECT_ROOT/configs/nginx/nginx.conf
touch $PROJECT_ROOT/configs/nginx/blue-upstream.conf
touch $PROJECT_ROOT/configs/nginx/green-upstream.conf
touch $PROJECT_ROOT/configs/nginx/templates/upstream.conf.template
touch $PROJECT_ROOT/configs/app/{blue.env,green.env,common.env}
touch $PROJECT_ROOT/configs/database/migrations.yaml

echo "Project structure created at $PROJECT_ROOT/"
```

Infrastructure Starter Code:

Here's complete, working code for a simple web application that will be deployed to blue and green environments. This is a prerequisite component that the deployment scripts will manage:

PYTHON

```python
# src/simple_web_app/app.py
"""
Simple web application with health endpoint for blue-green deployment.
Run with: python app.py --port PORT --color ENVIRONMENT_COLOR
"""

import os
import sys
import time
import json
import argparse
from http.server import HTTPServer, BaseHTTPRequestHandler
from datetime import datetime

class BlueGreenHandler(BaseHTTPRequestHandler):
    """HTTP request handler with health check and root endpoints."""

    def __init__(self, *args, **kwargs):
        self.environment_color = kwargs.pop('environment_color', 'unknown')
        self.start_time = time.time()
        self.version = os.getenv('APP_VERSION', '1.0.0')
        super().__init__(*args, **kwargs)

    def do_GET(self):
        """Handle GET requests."""
        if self.path == '/health':
            self.send_health_response()
        elif self.path == '/':
            self.send_root_response()
        else:
            self.send_error(404, "Not Found")

    def send_health_response(self):
        """Send health check response with status information."""
        status_data = {
            "status": "healthy",
            "environment": self.environment_color,
            "version": self.version,
            "uptime_seconds": round(time.time() - self.start_time, 2),
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "hostname": os.getenv('HOSTNAME', 'unknown'),
```

```
            "database_connected": self.check_database_connection()
        }

        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(status_data, indent=2).encode())

    def send_root_response(self):
        """Send root endpoint response."""
        response = {
            "message": f"Hello from {self.environment_color} environment!",
            "version": self.version,
            "endpoints": ["/", "/health"]
        }

        self.send_response(200)
        self.send_header('Content-Type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(response, indent=2).encode())

    def check_database_connection(self):
        """Check if database connection is working."""
        # Simulate database check - in real implementation, test actual DB connection
        db_url = os.getenv('DATABASE_URL', '')
        return bool(db_url and 'postgresql://' in db_url)

    def log_message(self, format, *args):
        """Override to log to stdout for containerized environments."""
        sys.stderr.write("%s - - [%s] %s\n" %
                         (self.address_string(),
                          self.log_date_time_string(),
                          format % args))

def run_server(port=8080, environment_color='blue'):
    """Start the HTTP server."""
    server_address = ('', port)

    # Create handler with custom arguments
    def handler(*args, **kwargs):
        BlueGreenHandler(*args, environment_color=environment_color, **kwargs)
```

```
    httpd = HTTPServer(server_address, handler)
    print(f"Starting {environment_color} server on port {port}...")
    print(f"Health check: http://localhost:{port}/health")
    httpd.serve_forever()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Blue-Green Deployment Web App')
    parser.add_argument('--port', type=int, default=int(os.getenv('PORT', 8080)),
                        help='Port to listen on')
    parser.add_argument('--color', default=os.getenv('ENVIRONMENT_COLOR', 'blue'),
                        help='Environment color (blue/green)')

    args = parser.parse_args()
    run_server(port=args.port, environment_color=args.color)
```

Core Logic Skeleton Code:

Here are the skeleton implementations for the main component entry points with detailed TODOs:

```bash
#!/bin/bash

# scripts/environment_manager/env_manager.sh - Environment Manager main script


# Source utilities

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

source "$SCRIPT_DIR/lib/utils.sh"


# Configuration

BLUE_PORT=8080

GREEN_PORT=8081

APP_DIR="$PROJECT_ROOT/src/simple_web_app"


# Main function to manage environment lifecycle

start_environment() {

    local env_color="$1"

    log_info "Starting $env_color environment..."



    # TODO 1: Determine port based on environment color

    #   - If env_color is "blue", use BLUE_PORT

    #   - If env_color is "green", use GREEN_PORT

    #   - Otherwise, log error and exit 1



    # TODO 2: Load environment-specific configuration

    #   - Source common.env from configs/app/

    #   - Source ${env_color}.env from configs/app/

    #   - Export all variables for the application



    # TODO 3: Check if environment is already running

    #   - Use lsof or netstat to check if port is in use

    #   - If running, log warning and return success (idempotent)



    # TODO 4: Start the application

    #   - Change to APP_DIR

    #   - Start python app.py with --port and --color arguments

    #   - Run in background, capture PID

    #   - Store PID in file for later management (e.g., /tmp/${env_color}_app.pid)



    # TODO 5: Wait for health check to pass

    #   - Call check_environment_health function

    #   - Retry up to 10 times with 2-second delays

    #   - If health check fails, stop the environment and exit with error
```

```
    log_success "$env_color environment started successfully on port $port"
}

deploy_to_environment() {
    local env_color="$1"
    local version="$2"
    log_info "Deploying version $version to $env_color environment..."

    # TODO 1: Validate inputs
    #   - env_color must be "blue" or "green"
    #   - version must match pattern vX.Y.Z or similar

    # TODO 2: Build the application artifact
    #   - This could be building a Docker image, compiling code, or packaging
    #   - Tag artifact with version and environment

    # TODO 3: Stop the existing environment if running
    #   - Call stop_environment function

    # TODO 4: Update environment configuration with new version
    #   - Set APP_VERSION=$version in environment config file
    #   - Update any other version-specific configuration

    # TODO 5: Start the environment with new version
    #   - Call start_environment function

    # TODO 6: Verify deployment
    #   - Call check_environment_health
    #   - Verify the version in health response matches deployed version

    log_success "Version $version deployed to $env_color environment"
}

# Other function skeletons with similar TODO structures...
# check_environment_health(), stop_environment(), get_environment_status()
```

```bash
#!/bin/bash

# scripts/load_balancer_controller/lb_controller.sh - Load Balancer Controller main script


# Configuration

ACTIVE_ENV_FILE="/tmp/active_environment"

NGINX_CONF_DIR="$PROJECT_ROOT/configs/nginx"

NGINX_MAIN_CONF="$NGINX_CONF_DIR/nginx.conf"


switch_traffic() {

    local target_env="$1"

    log_info "Switching traffic to $target_env environment..."



    # TODO 1: Validate target environment

    #   - Must be "blue" or "green"

    #   - Must be different from current active (optional but recommended)



    # TODO 2: Check target environment health

    #   - Call check_environment_health function from Environment Manager

    #   - If unhealthy, log error and exit 1



    # TODO 3: Generate new load balancer configuration

    #   - Call generate_load_balancer_config with target_env

    #   - This creates/updates upstream configuration files



    # TODO 4: Validate configuration syntax

    #   - Run nginx -t -c [config_path] to test configuration

    #   - If invalid, log error and exit 1



    # TODO 5: Perform graceful reload

    #   - Send nginx -s reload signal (not restart!)

    #   - Wait for reload to complete



    # TODO 6: Update active environment state

    #   - Write target_env to ACTIVE_ENV_FILE

    #   - Verify write was successful



    # TODO 7: Verify traffic is routing correctly

    #   - Make test request to load balancer

    #   - Check response contains target_env in headers or body

    #   - Retry up to 5 times with delay
```

```
    log_success "Traffic switched to $target_env environment"
}

get_inactive_environment() {
    log_info "Determining inactive environment..."

    # TODO 1: Read current active environment from ACTIVE_ENV_FILE
    #   - If file doesn't exist, default to "blue"

    # TODO 2: Return opposite environment
    #   - If active is "blue", return "green"
    #   - If active is "green", return "blue"
    #   - If active is neither, return "blue" as default

    # Note: This function should only return the color, not start/stop anything
}

# Other function skeletons: generate_load_balancer_config(), validate_config(), get_current_active()
```

Language-Specific Hints for Bash:

1. **Error Handling**: Use `set -euo pipefail` at the top of scripts to exit on errors, undefined variables, and pipeline failures. Trap signals for cleanup: `trap cleanup EXIT INT TERM`.
2. **Logging**: Create a logging function: `log_info() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] INFO: $*" | tee -a $LOG_FILE; }`. Use different levels (INFO, WARN, ERROR) with colors for readability.
3. **Configuration Management**: Source environment files safely: `set -a; source config.env; set +a`. Use `envsubst` for template substitution: `envsubst < template.conf > output.conf`.
4. **Process Management**: Use `pgrep -f` or `lsof -i:PORT` to check if processes are running. Store PIDs in files for management: `echo $PID > /tmp/app.pid`.
5. **Idempotency**: Design functions to be run multiple times safely. Check current state before taking action: `if is_environment_running "blue"; then restart_environment "blue"; else start_environment "blue"; fi`.
6. **Health Checks**: Use `curl` with timeout and retries: `curl --max-time 5 --retry 3 --retry-delay 2 --silent --fail http://localhost:$PORT/health`.

Milestone Checkpoint for High-Level Architecture:

After setting up the file structure and skeleton scripts, verify your architecture is correctly organized:

```bash
# Run from project root
cd blue-green-deployment

# Check directory structure
find . -type f -name "*.sh" | sort
# Expected output should show:
# ./scripts/environment_manager/env_manager.sh
# ./scripts/load_balancer_controller/lb_controller.sh
# ./scripts/deployment_orchestrator/deploy.sh
# ./scripts/database_migrator/migrator.sh

# Check key configuration files exist
ls -la configs/nginx/ configs/app/

# Test that the sample application runs
cd src/simple_web_app
python app.py --port 9999 --color test &
APP_PID=$!
sleep 2
curl -s http://localhost:9999/health | python -m json.tool
# Should show JSON with environment: "test", status: "healthy"
kill $APP_PID

echo "High-level architecture setup complete. Component skeletons are in place."
```

Debugging Tips for Architecture Issues:

Symptom	Likely Cause	How to Diagnose	Fix
"Script not found" errors when running from different directories	Relative paths in scripts not resolving correctly	Add `SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"` at top of each script and use absolute paths	Make all paths absolute or relative to `SCRIPT_DIR`
Environment variables not available in subprocesses	Variables not exported or sourced correctly	Add `export VAR_NAME` after setting variables. Use `env` command to see what's available	Source config files with `set -a; source file.env; set +a`
Multiple versions of scripts executing simultaneously	No lock mechanism for deployments	Check for existing deployment lock file	Implement file locking with `flock` or create lock file with PID
Component A can't call Component B's functions	Functions not available in subshell	Scripts are executed in separate processes, not sourced	Either source the script or redesign as standalone executables
Configuration changes not taking effect	Caching or not reloading services	Check if service needs reload vs restart	Use `nginx -s reload` not `systemctl restart nginx`

Data Model

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

The data model for a blue-green deployment system defines how configuration, state, and runtime information are structured and persisted. Think of this as the **DNA of your deployment infrastructure**—the genetic code that determines how each component behaves, how environments are distinguished, and how the system

remembers which version is currently serving users. Without a clear, consistent data model, your deployment system would be like a ship without navigation charts: components would operate with conflicting assumptions, state would become inconsistent, and the entire zero-downtime promise would collapse.

This section details two foundational aspects of the system's data model:

1. **Environment Variables & Configuration**: The runtime instructions that tell each environment who it is (blue or green), how to connect to its resources, and what version it should run.
2. **Deployment State**: The persistent memory that records which environment is currently active—the single source of truth for traffic routing decisions.

These elements work together like **name tags and a traffic controller's logbook** at a conference with two identical rooms. The name tags (environment configuration) tell each room whether it's "Room A" or "Room B," what time its session starts, and which speaker is presenting. The logbook (deployment state) tells the ushers which room currently has the main audience, so they know where to direct arriving attendees. Both must be accurate and synchronized for the conference to run smoothly.

Environment Variables & Configuration

In blue-green deployments, configuration management is critical because **blue and green must be identical twins with different identities**. They run the same application code but must be distinguishable (by color/port), connect to potentially different database instances or schemas during migrations, and report their status with accurate metadata. Configuration drift—where blue and green end up with different settings—is a silent killer of deployment reliability, leading to "it works in staging but not production" scenarios that undermine the entire strategy.

We use environment variables as the primary configuration mechanism because they're language-agnostic, easily injected at runtime, and work seamlessly with containerized deployments. Each environment receives its configuration at startup, which determines its identity and behavior throughout its lifecycle.

Core Environment Variables Table

Every environment (blue or green) requires these fundamental variables to establish its identity and operational parameters:

Variable Name	Example Value	Type	Description	Required For Milestone
`ENVIRONMENT_COLOR`	`"blue"` or `"green"`	String	The identity of this environment. Must match the deployment target color. Used by the application to tag logs, metrics, and health checks.	1
`APPLICATION_PORT`	`8080` (blue) or `8081` (green)	Integer	The port on which the application server listens. Each environment gets a distinct port to allow simultaneous operation.	1
`APP_VERSION`	`"v1.2.3"` or `"sha-abc123"`	String	The version identifier of the deployed artifact. Typically a Git commit SHA or semantic version. Reported in health checks.	1, 3
`DATABASE_URL`	`"postgresql://user:pass@localhost:5432/app_blue"`	String	Database connection string. May point to different database instances or schemas per environment to support expand-contract migrations.	1, 4
`FEATURE_FLAG_NEW_API`	`"true"` or `"false"`	Boolean	Example feature flag. Shows how environment-specific configuration can enable/disable features independently of code deployment.	1
`HEALTH_CHECK_PATH`	`"/health"`	String	The HTTP endpoint path for health checks. Consistent across environments but configurable.	1
`LOG_LEVEL`	`"INFO"`	String	Logging verbosity. Can differ between environments (e.g., more verbose in the inactive environment during testing).	1

> **Design Insight**: We separate identity (`ENVIRONMENT_COLOR`) from network location (`APPLICATION_PORT`) because while ports are the simplest distinction mechanism, future enhancements might use hostnames or container names. The color is the semantic identity; the port is the current implementation detail.

Application Configuration File Structure

While environment variables handle runtime injection, many applications also use configuration files. For consistency, we recommend a templated configuration file that gets populated with environment variables at deployment time. The structure below shows a typical `config.yaml` (or `config.json`) that the application reads:

`config.yaml` Template:

```yaml
# This file is generated during deployment by substituting environment variables
environment:
  color: "${ENVIRONMENT_COLOR}"
  port: "${APPLICATION_PORT}"
  version: "${APP_VERSION}"

database:
  url: "${DATABASE_URL}"
  pool_size: 20

features:
  new_api_enabled: ${FEATURE_FLAG_NEW_API}

logging:
  level: "${LOG_LEVEL}"
  health_endpoint: "${HEALTH_CHECK_PATH}"
```

The deployment process (specifically the `deploy_to_environment` function) would replace the `${VARIABLE}` placeholders with actual environment variable values before starting the application. This approach keeps configuration centralized and environment-specific without maintaining separate files for blue and green.

Health Check Response Data Structure

The health check endpoint (e.g., `GET /health`) returns a standardized JSON structure that provides runtime status and identity information. This is crucial for the load balancer to validate an environment's health before switching traffic and for operators to debug issues. The response must match the `BlueGreenHandler` structure defined in our naming conventions.

Health Check Response Fields:

Field Name	Type	Description	Example Value
`status`	String	Overall health status: `"healthy"`, `"unhealthy"`, or `"starting"`. The load balancer requires `"healthy"` to route traffic.	`"healthy"`
`environment`	String	The environment color: `"blue"` or `"green"`. Must match `ENVIRONMENT_COLOR`.	`"blue"`
`version`	String	Application version from `APP_VERSION`. Used to confirm correct deployment.	`"v1.2.3"`
`uptime_seconds`	Number	Seconds since application start. Helps identify freshly started environments.	`150`
`timestamp`	String	ISO 8601 timestamp of when the health check was generated.	`"2023-10-05T14:30:00Z"`
`hostname`	String	Hostname of the server/container. Useful for debugging in multi-instance setups.	`"web-blue-7f6b8c"`
`database_connected`	Boolean	Indicates successful database connection. Critical for validating database migrations.	`true`

This structure is returned by a dedicated health check handler (the `BlueGreenHandler` in our naming conventions) and provides a consistent interface for all monitoring and automation tools.

Database Configuration During Migrations

During database migrations (Milestone 4), the `DATABASE_URL` may differ between blue and green to support the expand-contract pattern. Consider a migration that adds a new column `new_field` to a `users` table:

Phase	Blue Environment `DATABASE_URL`	Green Environment `DATABASE_URL`	Purpose
Initial State	`postgresql://.../app_main`	`postgresql://.../app_main`	Both environments share the same database with schema v1.
Expand Phase	`postgresql://.../app_main` (v1)	`postgresql://.../app_main` (v1 with nullable column)	Green uses same DB but new code expects nullable `new_field`.
Contract Phase	`postgresql://.../app_main` (v2)	`postgresql://.../app_main` (v2)	After switch, both use v2 schema; old column may be dropped.

> **Critical Insight**: The expand-contract pattern requires **both application versions to work with the same database schema simultaneously** during the transition. Environment variables let us point blue and green to different database instances if needed, but more commonly we use a single database with backward-compatible schema changes.

Deployment State

The deployment state is the system's memory of which environment is currently active. This is a **single source of truth** that must be persistent, atomic, and consistently readable by all components. Think of it as the **air traffic control tower's display** showing which runway (blue or green) is currently open for landings. All decisions about where to route new traffic depend on this state.

State Storage Mechanism

We need a simple, reliable way to store and retrieve the active environment color. The state must survive process restarts and be accessible to multiple components (load balancer controller, deployment orchestrator, monitoring scripts).

> **Decision: State Storage Mechanism**
>
> - **Context**: We need a lightweight, persistent storage for the active environment color that is fast to read/write, requires minimal infrastructure, and works across different deployment environments (local development, cloud VMs, containers).
> - **Options Considered**:
>   1. **Plain text file** (e.g., `/tmp/active_environment`): Simple, no external dependencies, easy to read/write from shell scripts.
>   2. **Key-value store** (e.g., Redis, etcd): More robust for distributed systems, supports atomic operations, but adds infrastructure complexity.
>   3. **Database table**: Centralized and consistent, but overkill for a single value and adds query latency.
> - **Decision**: Use a plain text file with atomic write patterns for simplicity in this learning project.
> - **Rationale**: For educational purposes and simplicity, a file minimizes cognitive load and infrastructure dependencies. The atomicity requirement (preventing partial reads during writes) can be addressed with write-rename patterns. In production at scale, a distributed key-value store would be better, but here we prioritize understanding the core concepts.
> - **Consequences**: The state is local to the machine, which works for single-machine deployments. For multi-server setups, the file must be on shared storage or replaced with a distributed alternative. File-based locking may be needed if multiple processes write concurrently.

Comparison Table:

Option	Pros	Cons	Chosen?
Plain text file	Zero dependencies, extremely simple, fast, works everywhere.	Not atomic by default, local to filesystem, concurrency issues if not careful.	✅ Yes
Key-value store (Redis)	Atomic operations, distributed, supports watches/notifications.	Requires running Redis, network dependency, operational overhead.	❌ No
Database table	Consistent with application data, transactional guarantees.	Heavyweight, requires schema, slower than in-memory options.	❌ No

State File Format and Location

We store the active environment color in a plain text file at the location defined by the `ACTIVE_ENV_FILE` constant. The file contains exactly one line with either `"blue"` or `"green"` (no trailing whitespace).

Example state file contents:

```
blue
```

State File Operations:

Operation	Command/Logic	Notes
Read active environment	`cat $ACTIVE_ENV_FILE`	Must handle missing file (initial state). Default to `"blue"` if file doesn't exist.
Write active environment	Write to temporary file, then atomically rename: `echo "green" > ${ACTIVE_ENV_FILE}.tmp && mv ${ACTIVE_ENV_FILE}.tmp ${ACTIVE_ENV_FILE}`	The rename (`mv`) is atomic on POSIX systems, preventing partial reads.
Initialize if missing	`[-f "$ACTIVE_ENV_FILE"]	

State Transition Logic

The active environment state changes only during a traffic switch or rollback. The state machine is simple but must be handled carefully to prevent race conditions:

State Transition Table:

Current State	Event	Next State	Actions Taken
`"blue"`	`SWITCH_TO_GREEN` (green healthy)	`"green"`	1. Validate green health, 2. Update load balancer config, 3. Reload load balancer, 4. Write `"green"` to state file.
`"green"`	`SWITCH_TO_BLUE` (blue healthy)	`"blue"`	1. Validate blue health, 2. Update load balancer config, 3. Reload load balancer, 4. Write `"blue"` to state file.
`"green"`	`ROLLBACK` (after failed deployment)	`"blue"`	1. Immediately write `"blue"` to state file, 2. Update load balancer config, 3. Reload load balancer.
`"blue"`	`ROLLBACK` (after failed deployment)	`"green"`	1. Immediately write `"green"` to state file, 2. Update load balancer config, 3. Reload load balancer.

> **Design Principle**: **State updates follow action**—we only change the persistent state after successfully reconfiguring the load balancer. This ensures that if the system crashes during a switch, the state file still reflects the last successfully active environment, preventing traffic from being routed to an unprepared environment.

Determining Inactive Environment

A key function in the deployment orchestrator is `get_inactive_environment()`, which reads the state file and returns the opposite color. This tells the orchestrator where to deploy the new version without affecting live traffic.

Algorithm for `get_inactive_environment()`:

1. Read the current active environment from `ACTIVE_ENV_FILE`.
2. If the file doesn't exist or is empty, assume `"blue"` is active (and thus `"green"` is inactive).
3. If active is `"blue"`, return `"green"`.
4. If active is `"green"`, return `"blue"`.

This simple logic enables the core deployment flow: deploy to inactive, test, then switch.

Deployment Logging

Beyond the binary active/inactive state, we also maintain a deployment log for audit trails and debugging. The `DEPLOYMENT_LOG` constant defines a file where each deployment attempt is recorded with timestamp, version, target environment, and outcome.

Deployment Log Entry Format (CSV or JSON Lines):

```
timestamp,environment,version,action,success,message
2023-10-05T14:30:00Z,green,v1.2.3,deploy_started,true,Deploying to inactive environment
2023-10-05T14:32:00Z,green,v1.2.3,smoke_test_passed,true,All smoke tests passed
2023-10-05T14:33:00Z,green,v1.2.3,traffic_switch,true,Switched traffic to green
```

While not strictly required for the system to function, this logging is invaluable for troubleshooting and understanding deployment history.

Common Pitfalls

⚠️ Pitfall: Hardcoding ports in application code instead of using environment variables

- **Description**: Writing `app.listen(8080)` directly in your application code instead of reading from `APPLICATION_PORT`.
- **Why it's wrong**: Makes it impossible to run blue and green simultaneously on different ports. The application becomes inflexible and environment-aware code can't be reused.
- **Fix**: Always read ports, hostnames, and database URLs from environment variables. Use a configuration module that defaults to sane values but allows override.

⚠️ Pitfall: Storing state in memory without persistence

- **Description**: The load balancer controller keeps the active environment in a variable that disappears on restart.
- **Why it's wrong**: After a system reboot or process crash, the load balancer forgets which environment is active, potentially routing traffic incorrectly or requiring manual intervention.
- **Fix**: Always persist the active environment to disk (or a distributed store) after each switch, and read it on startup.

⚠️ Pitfall: Not validating configuration before deployment

- **Description**: Deploying a new version without checking that all required environment variables are set.
- **Why it's wrong**: The application may start but fail silently or behave incorrectly because of missing configuration, causing health checks to pass but functionality to be broken.
- **Fix**: Add a configuration validation step in `deploy_to_environment` that checks for required variables and validates their formats (e.g., port is a number, database URL is reachable).

⚠️ Pitfall: Using relative paths for state files

- **Description**: Setting `ACTIVE_ENV_FILE` to `"./active.env"` without considering the working directory.
- **Why it's wrong**: Different components (orchestrator, load balancer controller) may run from different directories, leading to multiple inconsistent state files.
- **Fix**: Use absolute paths for all state files, or define them relative to a known root directory (e.g., `/var/lib/bluegreen/`).

⚠️ Pitfall: Not handling missing state file on first run

- **Description**: The `get_inactive_environment()` function crashes when the state file doesn't exist.
- **Why it's wrong**: Breaks the deployment system on initial setup or after accidental deletion.
- **Fix**: Implement graceful defaults: if the state file is missing, assume `"blue"` is active (or create it with a default value).

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Configuration Management	Environment variables + shell scripts	Configuration management tool (Ansible, Puppet) or dedicated config service (HashiCorp Vault)
State Storage	Plain text file with atomic rename	Distributed key-value store (Redis, etcd) with watch capabilities
Configuration Templating	`envsubst` or `sed` in Bash	Dedicated templating engine (Jinja2, Go templates)

Recommended File Structure:

```
blue-green-deploy/
├── scripts/                    # All deployment and management scripts
│   ├── deploy.sh               # Main deployment orchestrator
│   ├── load-balancer/          # Load balancer management
│   │   ├── generate-config.sh
│   │   └── switch-traffic.sh
│   ├── environment/            # Environment management
│   │   ├── start-environment.sh
│   │   └── health-check.sh
│   └── migrations/             # Database migration scripts
│       ├── apply.sh
│       └── rollback.sh
├── config/                     # Configuration templates and defaults
│   ├── app-config.yaml.template
│   ├── nginx.conf.template
│   └── environment-defaults.sh
├── state/                      # State files (gitignored)
│   ├── active_environment
│   └── deployment.log
├── src/                        # Application source code
│   ├── app.py                  # Sample application
│   └── health_handler.py       # Health check endpoint implementation
└── .env.example                # Example environment variables
```

Infrastructure Starter Code:

Here's a complete Bash script to manage environment variables and configuration generation. Save this as `scripts/config-helper.sh`:

```bash
#!/bin/bash

set -euo pipefail


# Load configuration defaults

CONFIG_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")/../config" && pwd)"

source "${CONFIG_DIR}/environment-defaults.sh"


# Function: validate required environment variables

validate_config() {

    local required_vars=("ENVIRONMENT_COLOR" "APPLICATION_PORT" "APP_VERSION" "DATABASE_URL")

    for var in "${required_vars[@]}"; do

        if [[ -z "${!var:-}" ]]; then

            echo "ERROR: Required environment variable $var is not set"

            return 1

        fi

    done



    # Validate ENVIRONMENT_COLOR

    if [[ "$ENVIRONMENT_COLOR" != "blue" && "$ENVIRONMENT_COLOR" != "green" ]]; then

        echo "ERROR: ENVIRONMENT_COLOR must be 'blue' or 'green', got '$ENVIRONMENT_COLOR'"

        return 1

    fi



    # Validate APPLICATION_PORT is a number

    if ! [[ "$APPLICATION_PORT" =~ ^[0-9]+$ ]]; then

        echo "ERROR: APPLICATION_PORT must be a number, got '$APPLICATION_PORT'"

        return 1

    fi



    return 0

}


# Function: generate application config from template

generate_app_config() {

    local template_file="${CONFIG_DIR}/app-config.yaml.template"

    local output_file="/tmp/app-config-${ENVIRONMENT_COLOR}.yaml"



    if [[ ! -f "$template_file" ]]; then

        echo "ERROR: Template file not found: $template_file"

        return 1

    fi
```

```
    # Replace all ${VAR} with environment variable values
    envsubst < "$template_file" > "$output_file"

    echo "Generated config at $output_file"
    echo "--- Config content ---"
    cat "$output_file"
    echo "--- End config ---"

    echo "$output_file"
}

# Function: get inactive environment
get_inactive_environment() {
    local state_file="${STATE_DIR:-/tmp}/active_environment"
    local active_env="blue"

    if [[ -f "$state_file" ]]; then
        active_env=$(cat "$state_file" | tr -d '[:space:]')
    fi

    if [[ "$active_env" == "blue" ]]; then
        echo "green"
    else
        echo "blue"
    fi
}

# Function: read deployment state safely
read_active_environment() {
    local state_file="${STATE_DIR:-/tmp}/active_environment"

    if [[ ! -f "$state_file" ]]; then
        # Initialize with default
        echo "blue" > "$state_file"
        echo "blue"
        return
    fi

    local content
    content=$(cat "$state_file" | tr -d '[:space:]')
```

```
    if [[ "$content" != "blue" && "$content" != "green" ]]; then
        echo "WARNING: Invalid content in state file, resetting to 'blue'" >&2
        echo "blue" > "$state_file"
        echo "blue"
    else
        echo "$content"
    fi
}

# Function: write deployment state atomically
write_active_environment() {
    local new_active="$1"
    local state_file="${STATE_DIR:-/tmp}/active_environment"

    if [[ "$new_active" != "blue" && "$new_active" != "green" ]]; then
        echo "ERROR: Cannot write invalid environment '$new_active' to state file"
        return 1
    fi

    # Atomic write using temporary file and rename
    local temp_file="${state_file}.tmp.$$"
    echo "$new_active" > "$temp_file"
    mv "$temp_file" "$state_file"

    echo "Updated active environment to: $new_active"
}
```

Core Logic Skeleton Code:

Here's the skeleton for the main deployment state management functions to be implemented in your deployment orchestrator:

```bash
#!/bin/bash

# File: scripts/deploy-state.sh

# TODO: Integrate this with the main deployment orchestrator


# Constants (use exact names from conventions)

ACTIVE_ENV_FILE="/tmp/active_environment"

DEPLOYMENT_LOG="/var/log/deployments.log"


# Function: get_inactive_environment() returns string

# Determines which environment is not currently active

get_inactive_environment() {

    # TODO 1: Read the active environment from ACTIVE_ENV_FILE

    # TODO 2: If file doesn't exist or is empty, default to "blue" as active

    # TODO 3: If active is "blue", return "green"

    # TODO 4: If active is "green", return "blue"

    # TODO 5: Log the result for debugging

    # Hint: Use the read_active_environment function from config-helper.sh

}


# Function: switch_traffic(target_env) returns void

# Updates load balancer to route to target environment

switch_traffic() {

    local target_env="$1"



    # TODO 1: Validate target_env is "blue" or "green"

    # TODO 2: Check health of target environment (call check_health function)

    # TODO 3: Generate new load balancer configuration for target_env

    # TODO 4: Validate configuration syntax (e.g., nginx -t)

    # TODO 5: Reload load balancer gracefully (e.g., nginx -s reload)

    # TODO 6: Update state file atomically with write_active_environment

    # TODO 7: Log the traffic switch to DEPLOYMENT_LOG

    # TODO 8: Verify traffic is actually flowing to target environment

}


# Function: rollback_deployment(failed_env) returns void

# Reverts traffic to previous environment on failure

rollback_deployment() {

    local failed_env="$1"



    # TODO 1: Determine previous environment (opposite of failed_env)

    # TODO 2: Check health of previous environment

    # TODO 3: Update state file immediately to previous environment (safety first)
```

```
    # TODO 4: Generate load balancer config for previous environment
    # TODO 5: Reload load balancer
    # TODO 6: Log rollback with details to DEPLOYMENT_LOG
    # TODO 7: Send alert/notification about rollback
    # TODO 8: Mark failed environment for investigation
}
```

Language-Specific Hints (Bash):

- Use `set -euo pipefail` at the top of scripts for strict error handling.
- For atomic file writes, create a temporary file and use `mv` (rename), which is atomic on POSIX systems.
- Source configuration files with `source file.sh` or `. file.sh`.
- Validate environment variables with parameter expansion: `"${VAR:?Error message}"` will exit if VAR is unset.
- Use `envsubst` from the `gettext` package for template substitution: `sudo apt-get install gettext`.
- Log with timestamps: `echo "$(date -Iseconds): Message" >> $DEPLOYMENT_LOG`.

Milestone Checkpoint: After implementing the data model, run these verification commands:

```bash
# 1. Test configuration validation
export ENVIRONMENT_COLOR="blue"
export APPLICATION_PORT="8080"
export APP_VERSION="v1.0.0"
export DATABASE_URL="postgresql://test@localhost/test"
./scripts/config-helper.sh validate_config
# Expected: No error output, exit code 0

# 2. Test state management
unset ACTIVE_ENV_FILE  # Use default
./scripts/deploy-state.sh get_inactive_environment
# Expected: "green" (since default active is blue)

# 3. Test state file creation and reading
rm -f /tmp/active_environment
./scripts/config-helper.sh read_active_environment
# Expected: Creates file with "blue", prints "blue"

# 4. Test atomic write
./scripts/config-helper.sh write_active_environment "green"
cat /tmp/active_environment
# Expected: File contains exactly "green" (no newline issues)

# 5. Verify health check structure
curl -s http://localhost:8080/health | jq .
# Expected: JSON with status, environment, version, uptime_seconds, etc.
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Application starts on wrong port	`APPLICATION_PORT` not set or incorrect	`echo $APPLICATION_PORT` in startup script	Ensure port is set before starting app
State file resets to blue on every run	File permissions or script running from different directory	`ls -la /tmp/active_environment`, check file path consistency	Use absolute path for `ACTIVE_ENV_FILE`
Health check returns wrong environment color	`ENVIRONMENT_COLOR` variable not passed to application	Check environment variables in container/process: `ps aux | grep app`	Pass all required variables at runtime
Traffic switch doesn't persist after restart	State only in memory, not written to disk	Check if `write_active_environment` is called after switch	Ensure state file is written atomically after successful switch
Both environments show as active	State file corrupted or contains invalid value	`cat $ACTIVE_ENV_FILE`, check for whitespace or invalid strings	Implement validation in `read_active_environment`

Milestone(s): Milestone 1 (Dual Environment Setup)

Component: Environment Manager

The Environment Manager is the foundational pillar of the blue-green deployment architecture. It is responsible for creating, configuring, and maintaining the two independent production environments—**blue** and **green**. Its primary design goal is to treat these environments as completely isolated, identical twins, ensuring that deploying a new version to one has absolutely no effect on the other, which may be actively serving live user traffic at that moment.

Mental Model: Identical Twins

Imagine you have identical twin siblings, Blue and Green. They are genetically identical (same application codebase) but are individuals living in separate, self-contained apartments (isolated runtime environments). Each twin's apartment has its own address (port or hostname), its own set of house keys (environment variables), and its own independent utilities (database connections, caches). You can visit one twin to deliver a package (deploy new code) without disturbing the other twin who might be hosting a party (serving live traffic). The twins are so similar that if you swapped their apartments, guests (user requests) wouldn't notice the difference—except perhaps for a different paint color on the walls (environment identity).

This mental model clarifies three core principles:

1. **Isolation:** Blue and Green must run in separate, non-interfering runtime contexts. A crash, configuration error, or resource exhaustion in one must not affect the other.
2. **Identity:** Each environment must have a clear, immutable identifier (`BLUE` or `GREEN`) that is known to the application, the deployment system, and the load balancer. This identity dictates its operational parameters.
3. **Idempotency:** The act of setting up or deploying to an environment should be repeatable and produce the same result every time. Whether the environment is being created for the first time or being updated, the outcome is a correctly configured, running instance.

Interface and Responsibilities

The Environment Manager exposes a set of operations that allow other components (primarily the Deployment Orchestrator) to control the lifecycle of each environment. Its interface is defined by the following functions and data structure.

Core Data Structure: `BlueGreenHandler`

This data structure represents the standardized health and status information returned by each environment's health check endpoint. It provides a unified view for the load balancer and deployment system to assess an environment's readiness.

Field Name	Type	Description
`status`	`string`	Overall health status. Must be `"healthy"` for the environment to receive traffic. Other values like `"degraded"` or `"unhealthy"` indicate problems.
`environment`	`string`	The immutable identity of the environment: `"blue"` or `"green"`.
`version`	`string`	The version identifier of the application currently deployed (e.g., git commit hash, semantic version).
`uptime_seconds`	`number`	How long the application process has been running, in seconds. Useful for detecting recent restarts.
`timestamp`	`string`	ISO-8601 timestamp of when the status was generated.
`hostname`	`string`	The hostname of the server where the environment is running. Helps with debugging in multi-server setups.
`database_connected`	`boolean`	Indicates whether the application can successfully connect to its configured database. A critical readiness check.

Interface Methods

The following table details the functions the Environment Manager component must implement. These are the commands the "conductor" (Deployment Orchestrator) uses to manage the "twins."

Method Name	Parameters	Returns	Description
`start_environment(env_color)`	`env_color` (`string`): The color identifier, `"blue"` or `"green"`.	`void`	**Bootstraps the environment.** This function is responsible for provisioning the runtime environment for the specified color. Its duties include: setting environment-specific variables (e.g., `PORT`, `DATABASE_URL`), launching the application process (e.g., a Docker container or a systemd service), and waiting until the application's health endpoint reports `"healthy"`. It must be idempotent; calling it on an already-running environment should have no negative effect.
`stop_environment(env_color)`	`env_color` (`string`): The color identifier, `"blue"` or `"green"`.	`void`	**Gracefully shuts down the environment.** Stops the application process and releases any allocated resources (e.g., network ports, container instances). The shutdown should allow in-flight requests to complete (graceful shutdown) to prevent user errors during a traffic switch or rollback.
`check_health(env_color)`	`env_color` (`string`): The color identifier, `"blue"` or `"green"`.	`boolean`	**Performs a deep health assessment.** This is more than checking if the process is running. It queries the environment's health check endpoint (which returns a `BlueGreenHandler` JSON object) and validates the response. It returns `true` only if: the HTTP request succeeds, the `status` field equals `"healthy"`, and critical dependencies like `database_connected` are `true`. This function is the gatekeeper before any traffic switch.
`deploy_to_environment(env_color, version)`	`env_color` (`string`): Target environment color. `version` (`string`): The new application version to deploy.	`void`	**Orchestrates a deployment to a specific environment.** This is a higher-level operation that may call `stop_environment`, build/retrieve the new version artifact, inject configuration, and then call `start_environment`. Its key responsibility is to ensure the new version is running and healthy in the target environment, ready to take traffic.

Architecture Decision: Environment Identity

> **Decision: Use Distinct TCP Ports for Environment Identity**
>
> - **Context**: We need a simple, foolproof way to run two identical copies of the same application on the same host(s) during blue-green deployments. The environments must be distinguishable by the load balancer and the deployment scripts.
> - **Options Considered**:
>   1. **Different TCP Ports**: Run the Blue environment on port `BLUE_PORT` (e.g., 8080) and the Green environment on port `GREEN_PORT` (e.g., 8081).
>   2. **Different Hostnames/Subdomains**: Run both environments on the same port but different hostnames (e.g., `blue.app.example.com`, `green.app.example.com`).
>   3. **Container Identifiers**: Run each environment in separate Docker containers or Kubernetes pods, identified by container names or labels.
> - **Decision**: We chose **Option 1 (Different TCP Ports)** for the initial implementation.
> - **Rationale**:
>   - **Simplicity**: It requires minimal infrastructure. No DNS changes, no complex container orchestration. The application only needs to bind to a port configurable via an environment variable.
>   - **Local Development Friendly**: Developers can easily run both environments side-by-side on their laptops for testing.
>   - **Explicit Control**: The port number is a concrete, unambiguous identifier for the load balancer (Nginx/HAProxy) configuration.
>   - **Low Overhead**: It works on a single server, making it ideal for learning and small-scale deployments.
> - **Consequences**:
>   - **Port Management**: The system must ensure the ports are not used by other processes. Scripts must explicitly reference `BLUE_PORT` and `GREEN_PORT`.
>   - **Not Suitable for Horizontal Scaling**: This model assumes one instance per environment per host. Scaling out (multiple servers) would require a more sophisticated approach (like Option 2 or 3).
>   - **Firewall Rules**: If running on cloud VMs, security groups must allow traffic on both ports.

Option	Pros	Cons	Chosen?
Different TCP Ports	Extremely simple, works anywhere, easy to debug.	Doesn't scale horizontally well. Ports are a limited resource.	**Yes**
Different Hostnames	Allows identical port usage. Scales well with DNS or service discovery.	Requires DNS setup or `/etc/hosts` modification. Slightly more complex.	No
Container Identifiers	Excellent isolation, industry standard (K8s, Docker), ideal for scaling.	Requires container runtime expertise. Adds operational complexity.	No

Common Pitfalls

⚠️ Pitfall: Configuration Drift

- **Description**: Manually changing environment variables, files, or dependencies in the Blue environment but forgetting to apply the same changes to Green (or vice-versa). Over time, the environments become non-identical, breaking the core assumption of the blue-green pattern.
- **Why it's Wrong**: A deployment to the drifted environment may fail or behave unexpectedly. A traffic switch to a misconfigured environment could cause a production outage.
- **How to Fix**: **Inject all configuration via environment variables or a centralized config service at startup.** Never rely on manual, out-of-band changes. Use Infrastructure as Code (IaC) tools (e.g., Terraform, Ansible) to provision both environments from the same definition.

⚠️ Pitfall: Shared Resources

- **Description**: Allowing Blue and Green environments to share a resource that is not designed for concurrent, version-independent access. The most common example is a database schema that is not backward-compatible.
- **Why it's Wrong**: Deploying a new version with breaking database changes to the inactive environment will immediately break the active environment that is still running the old code and expects the old schema.
- **How to Fix**: **Treat the database as part of the deployed artifact.** Use the expand-contract migration pattern (detailed in the Database Migrator component) to ensure database schema changes are backward-compatible. Both old (active) and new (inactive) application versions must work correctly with the current database schema.

⚠️ Pitfall: Inadequate Health Checks

- **Description**: Implementing a health check endpoint that only checks if the web server process is alive (a "liveness" probe), without verifying critical downstream dependencies like the database, cache, or external APIs.
- **Why it's Wrong**: The load balancer might switch traffic to an environment that is running but unable to process requests correctly, leading to user-facing errors.
- **How to Fix**: **Implement a comprehensive "readiness" probe.** The health check must validate all essential dependencies. The `BlueGreenHandler` structure includes a `database_connected` field for this purpose. The `check_health()` function must fail if any critical dependency is unhealthy.

Implementation Guidance

This section provides concrete starter code to build the Environment Manager. The primary language is Bash, with a simple Python web application for demonstration.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Application Server	Python Flask (single-file, easy)	Go (binary, performant) or Docker containers
Process Management	Bash scripts with `&` and `pkill`	Systemd service units or Docker Compose
Health Check	HTTP endpoint returning JSON	Same, but with more detailed dependency checks (e.g., cache, message queue)

B. Recommended File/Module Structure

Place the environment management scripts and application code in a well-organized directory. This mirrors the High-Level Architecture.

```
project-root/
├── bin/
│   ├── deploy.sh                # Deployment Orchestrator (Milestone 3)
│   ├── switch-traffic.sh        # Load Balancer Controller (Milestone 2)
│   └── rollback.sh              # Rollback automation
├── environments/
│   ├── blue/                    # Blue environment runtime directory
│   │   ├── app.py               # The application code (symlink or copy)
│   │   ├── .env                 # Blue-specific environment variables
│   │   └── pidfile              # Stores the Blue app's process ID
│   ├── green/                   # Green environment runtime directory
│   │   ├── app.py
│   │   ├── .env
│   │   └── pidfile
│   └── manager.sh               # Environment Manager functions (start, stop, check)
├── migrations/                  # Database Migrator scripts (Milestone 4)
├── config/
│   └── nginx.conf.template      # Load balancer config template
└── README.md
```

C. Infrastructure Starter Code

Here is a complete, minimal web application in Python that implements the required health check endpoint and serves as the application to be deployed. Save this as `environments/app.py`.

PYTHON

```python
#!/usr/bin/env python3
"""
Blue-Green Deployment Sample Application.
Serves a simple API and a health check endpoint conforming to BlueGreenHandler.
"""
import json
import os
import time
from datetime import datetime
from flask import Flask, jsonify

app = Flask(__name__)

# Environment identity and configuration injected at startup
ENVIRONMENT = os.getenv('ENVIRONMENT', 'unknown')
APP_VERSION = os.getenv('APP_VERSION', 'v0.0.0')
PORT = int(os.getenv('PORT', 8080))
DATABASE_URL = os.getenv('DATABASE_URL', '')
# Simulate a database connection check
DATABASE_CONNECTED = False

@app.before_first_request
def init_app():
    """Simulate a database connection check on startup."""
    global DATABASE_CONNECTED, START_TIME
    START_TIME = time.time()
    # In a real app, try to connect to DATABASE_URL here
    if DATABASE_URL:
        DATABASE_CONNECTED = True

@app.route('/')
def index():
    """Main endpoint returns environment info."""
    return jsonify({
        'message': f'Hello from {ENVIRONMENT} environment',
        'version': APP_VERSION,
        'environment': ENVIRONMENT
    })

@app.route('/api/health')
def health():
    """Health check endpoint returning BlueGreenHandler JSON."""
    uptime = time.time() - START_TIME
```

```
    status = "healthy" if DATABASE_CONNECTED else "unhealthy"
    return jsonify({
        'status': status,
        'environment': ENVIRONMENT,
        'version': APP_VERSION,
        'uptime_seconds': round(uptime, 2),
        'timestamp': datetime.utcnow().isoformat() + 'Z',
        'hostname': os.uname().nodename,
        'database_connected': DATABASE_CONNECTED
    })

if __name__ == '__main__':
    init_app()
    print(f"Starting {ENVIRONMENT} server on port {PORT}, version {APP_VERSION}")
    app.run(host='0.0.0.0', port=PORT)
```

D. Core Logic Skeleton Code

The main Environment Manager script, `environments/manager.sh`, implements the interface functions. Below is the skeleton with detailed TODOs.

```bash
#!/usr/bin/env bash

# Environment Manager for Blue-Green Deployments

# Usage: ./manager.sh [start|stop|check|deploy] [blue|green] [version]


set -euo pipefail  # Strict error handling


# --- Configuration Constants ---

readonly BLUE_PORT="8080"

readonly GREEN_PORT="8081"

readonly PROJECT_ROOT="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd)"

readonly ACTIVE_ENV_FILE="/tmp/active_environment"

readonly DEPLOYMENT_LOG="/var/log/deployments.log"


# --- Helper Functions ---

log() {

    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$DEPLOYMENT_LOG"

}


get_port_for_environment() {

    local env_color="$1"

    # TODO 1: Return the port number for the given environment color.

    # Hint: Use a case statement on $env_color to echo $BLUE_PORT or $GREEN_PORT.

}


get_pidfile_for_environment() {

    local env_color="$1"

    echo "$PROJECT_ROOT/environments/$env_color/pidfile"

}


is_environment_running() {

    local env_color="$1"

    local pidfile

    pidfile=$(get_pidfile_for_environment "$env_color")

    # TODO 2: Check if the pidfile exists and if the process ID within it is still running.

    # Hint: Use `[ -f "$pidfile" ]` and `kill -0` on the stored PID.

}


# --- Core Interface Functions ---

start_environment() {

    local env_color="$1"

    local port

    port=$(get_port_for_environment "$env_color")

    log "Starting $env_color environment on port $port"
```

```
    # TODO 3: Set environment-specific variables.
    # 1. Create the environment directory if it doesn't exist.
    # 2. Create a .env file in the environment directory with:
    #    ENVIRONMENT=$env_color
    #    PORT=$port
    #    APP_VERSION=$(cat version.txt 2>/dev/null || echo "unknown")  # Version from a file
    #    DATABASE_URL="postgresql://user:pass@localhost/db_$env_color"  # Example

    # TODO 4: Launch the application process.
    # 1. Change directory to the environment's folder.
    # 2. Start the Python app (app.py) in the background, redirecting output to a log file.
    # 3. Capture the process ID (PID) and write it to the pidfile.

    # TODO 5: Wait for the health check to pass.
    # 1. Use a loop (with a timeout) to curl the health endpoint at http://localhost:$port/api/health.
    # 2. Parse the JSON response to ensure "status" is "healthy".
    # 3. Exit with an error if the health check does not pass within the timeout.

    log "$env_color environment started successfully"
}

stop_environment() {
    local env_color="$1"
    log "Stopping $env_color environment"

    # TODO 6: Gracefully stop the environment.
    # 1. Check if the environment is running using is_environment_running.
    # 2. If running, send a SIGTERM signal to the process ID from the pidfile.
    # 3. Wait for the process to exit (with a timeout).
    # 4. If it doesn't exit, send SIGKILL.
    # 5. Remove the pidfile.

    log "$env_color environment stopped"
}

check_health() {
    local env_color="$1"
    local port
    port=$(get_port_for_environment "$env_color")
    log "Checking health of $env_color environment on port $port"

    # TODO 7: Perform a comprehensive health check.
    # 1. Use curl to fetch http://localhost:$port/api/health with a timeout.
    # 2. Check the HTTP response code.
```

```
    # 3. Parse the JSON response using a tool like `jq` or Python.
    # 4. Validate that the "status" field equals "healthy" AND "database_connected" is true.
    # 5. Return 0 (success) if healthy, 1 (failure) otherwise.

    # Placeholder: Assume unhealthy if this function hasn't been implemented.
    return 1
}

deploy_to_environment() {
    local env_color="$1"
    local version="$2"
    log "Deploying version $version to $env_color environment"

    # TODO 8: Orchestrate a deployment.
    # 1. Stop the target environment (call stop_environment).
    # 2. Update the version file (e.g., echo "$version" > version.txt) in the environment directory.
    # 3. Start the target environment (call start_environment). It will pick up the new version.
    # 4. Verify the deployment by checking the health and that the reported version matches $version.

    log "Deployed version $version to $env_color successfully"
}

# --- Main Script Dispatcher ---
main() {
    local command="$1"
    local environment="$2"
    local version="${3:-}"

    case "$command" in
        start)
            start_environment "$environment"
            ;;
        stop)
            stop_environment "$environment"
            ;;
        check)
            if check_health "$environment"; then
                echo "HEALTHY"
                exit 0
            else
                echo "UNHEALTHY"
                exit 1
            fi
```

```
            ;;
        deploy)
            if [ -z "$version" ]; then
                echo "Usage: $0 deploy <blue|green> <version>"
                exit 1
            fi
            deploy_to_environment "$environment" "$version"
            ;;
        *)
            echo "Unknown command: $command"
            exit 1
            ;;
    esac
}

if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    if [ $# -lt 2 ]; then
        echo "Usage: $0 [start|stop|check|deploy] [blue|green] [version]"
        exit 1
    fi
    main "$@"
fi
```

E. Language-Specific Hints

- **Process Management in Bash**: Use `$!` to get the PID of the last backgrounded command. Always store it immediately. `kill -0 $PID` checks if a process exists without sending a signal.
- **JSON Parsing in Bash**: Install `jq`. Use `curl -s http://localhost:$PORT/api/health | jq -r .status` to extract the status field. For simple checks, `grep` can work but is fragile.
- **Timeouts**: Use the `timeout` command (e.g., `timeout 30 curl ...`) to prevent health check loops from hanging indefinitely.
- **Graceful Shutdown**: Your application (the Python Flask app) must catch SIGTERM and shut down cleanly. Flask's development server does this by default. In production, use a WSGI server like Gunicorn that supports graceful shutdown.

F. Milestone Checkpoint

After implementing the TODOs in `manager.sh` and creating the Python app, you should be able to complete Milestone 1: Dual Environment Setup.

Verification Commands:

```bash
# Make scripts executable
chmod +x environments/manager.sh environments/app.py

# Start the Blue environment
./environments/manager.sh start blue

# Start the Green environment
./environments/manager.sh start green

# Check their health independently
./environments/manager.sh check blue  # Should output "HEALTHY"
./environments/manager.sh check green # Should output "HEALTHY"

# Manually test the isolation: Deploy a new version to Green while Blue is active.
echo "v1.1.0" > environments/green/version.txt
./environments/manager.sh deploy green v1.1.0

# Verify Blue is still running the old version
curl http://localhost:8080/ | jq .version  # Should be "unknown" or old version
curl http://localhost:8081/ | jq .version  # Should be "v1.1.0"
```

Expected Behavior:

- Both environments run simultaneously on ports 8080 and 8081.
- The `/api/health` endpoint returns a JSON object matching the `BlueGreenHandler` structure, including the correct `environment` and `version` fields.
- Deploying to Green does not affect Blue's operation, and both remain healthy.

Signs of Trouble:

- **"Address already in use"**: Another process is using port 8080 or 8081. Use `lsof -i :8080` to find and stop it.
- **Health check fails**: Check the application log in the environment directory. Ensure the DATABASE_URL is set (even a dummy value) so `database_connected` can become true.
- **Script errors on `jq`**: Install jq with `apt-get install jq` or `brew install jq`.

> **Milestone(s):** Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

Component: Load Balancer Controller

The **Load Balancer Controller** is the traffic cop of the blue-green system. Its core responsibility is to manage the flow of all user requests, directing them to the currently active production environment while providing a mechanism for atomic, zero-downtime switching between the blue and green environments. This component embodies the principle of **separation of concerns**: the application code runs in the environments, while the controller decides which environment serves live traffic, enabling clean, independent management of each.

Mental Model: Railroad Switch Operator

Imagine a busy railway station with two parallel tracks: Track Blue and Track Green. A single stream of trains (user requests) arrives at the station and must be directed onto one of these tracks. The **railroad switch operator** (our Load Balancer Controller) sits at a control panel. Normally, all trains are routed onto Track Blue, which is fully operational. When maintenance or an upgrade is needed on Track Blue, the operator first ensures Track Green is prepared, inspected, and ready to carry traffic. Then, with a single, swift lever pull (the `switch_traffic` command), the switch points are reconfigured. The very next train—and all subsequent trains—are now seamlessly routed onto Track Green. Track Blue can now be taken offline for work without a single train being delayed or derailed. This atomic lever pull, preceded by a safety check, is the essence of zero-downtime switching.

Interface and Responsibilities

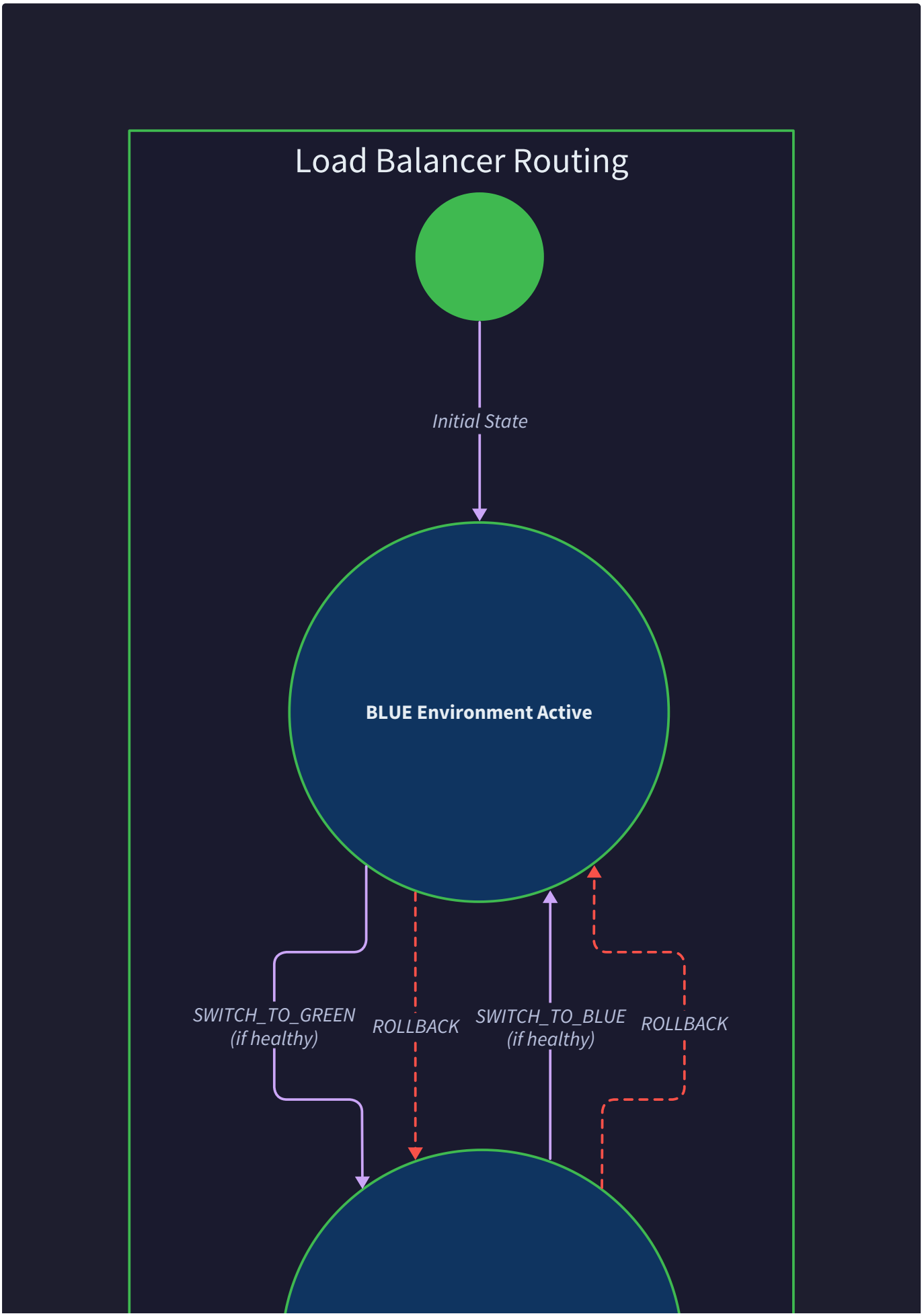
The Load Balancer Controller exposes a minimal, focused interface centered on state management and atomic transitions. Its primary data structure is the active environment state, persistently stored in `ACTIVE_ENV_FILE`. Its methods are designed to be idempotent and safe to retry.

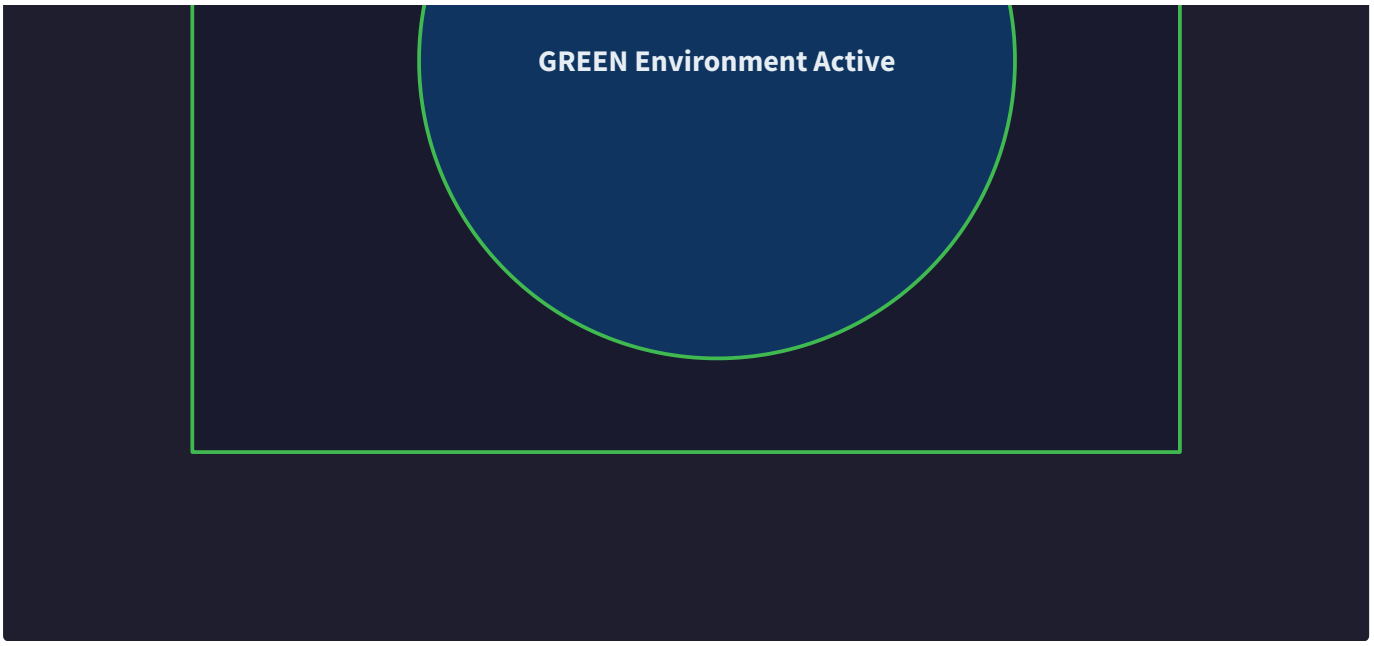
Method Name	Parameters	Returns	Description
`switch_traffic(target_env)`	`target_env`: String (`"blue"` or `"green"`)	`void`	Atomically updates the load balancer configuration to route all incoming traffic to the specified `target_env`. This operation must include a health validation check on the target environment and use a graceful reload mechanism to preserve in-flight connections. This is the core action for both deployment cutover and rollback.
`get_current_active_env()`	None	`String` (`"blue"`, `"green"`, or `"unknown"`)	Reads and returns the currently active environment color from the persistent state file (`ACTIVE_ENV_FILE`). This is a read-only operation used by other components (like the Deployment Orchestrator) to determine the deployment target.
`validate_load_balancer_config()`	None	`boolean`	Validates the syntax and semantics of the generated load balancer configuration file (e.g., `nginx.conf`). This is a critical pre-flight check to prevent a faulty configuration from taking down the service during a reload.
`generate_config_for_env(env_color)`	`env_color`: String (`"blue"` or `"green"`)	`String` (config content)	Generates the load balancer configuration snippet (e.g., an `upstream` block for Nginx) for a specific environment, using its predefined port (`BLUE_PORT`, `GREEN_PORT`) and host.

The component's internal state is simple but critical:

State Variable	Type	Storage	Description
`current_active_env`	`String` (`"blue"` or `"green"`)	File: `ACTIVE_ENV_FILE` (e.g., `/tmp/active_environment`)	The single source of truth for which environment (blue or green) is currently receiving live user traffic. This must be persisted to survive process restarts and to provide a consistent view to all system components.

The state transitions are visualized in the following diagram, where the switch is only allowed if the target environment is verified as healthy.


Load Balancer Routing
Initial State
BLUE Environment Active
SWITCH_TO_GREEN
(if healthy)
ROLLBACK
SWITCH_TO_BLUE
(if healthy)
ROLLBACK



The transition logic can be described in a state table:

Current State	Event/Condition	Next State	Actions Taken
`BLUE_ACTIVE`	`switch_traffic("green")` called AND `check_health("green")` passes	`GREEN_ACTIVE`	1. Generate config with green as primary upstream. 2. Validate new config. 3. Gracefully reload load balancer. 4. Update `ACTIVE_ENV_FILE` to `"green"`.
`GREEN_ACTIVE`	`switch_traffic("blue")` called AND `check_health("blue")` passes	`BLUE_ACTIVE`	1. Generate config with blue as primary upstream. 2. Validate new config. 3. Gracefully reload load balancer. 4. Update `ACTIVE_ENV_FILE` to `"blue"`.
`BLUE_ACTIVE`	`rollback_deployment()` called (e.g., after failed green deploy)	`BLUE_ACTIVE` (no change)	Traffic remains on blue. Log rollback attempt.
`GREEN_ACTIVE`	`rollback_deployment()` called (e.g., after failed blue deploy)	`GREEN_ACTIVE` (no change)	Traffic remains on green. Log rollback attempt.
`BLUE_ACTIVE` / `GREEN_ACTIVE`	`switch_traffic(target)` called BUT `check_health(target)` fails	No change	Abort switch. Log health check failure. Raise error to orchestrator.

Architecture Decision: Traffic Switching Mechanism

Decision: Load Balancer Configuration Reload for Traffic Switching

- **Context**: We need an atomic, fast, and reliable method to shift all user traffic from one environment to another. The mechanism must support instant rollback, preserve in-flight connections, and integrate with health checks. The solution must be operable within a typical virtualized/cloud infrastructure without requiring specialized networking hardware.
- **Options Considered**:
  1. **DNS Record Updates**: Changing the DNS A/AAAA record for the service hostname (e.g., `app.example.com`) to point to the IP address of the new environment.
  2. **Load Balancer Config Update & Reload**: Using a reverse proxy (Nginx, HAProxy) with a configuration file that defines upstream servers. Switching involves updating this file to point to the new environment's upstream block and issuing a graceful reload command.
  3. **Service Mesh Traffic Splitting**: Using a service mesh (e.g., Istio, Linkerd) to define traffic routing rules (e.g., VirtualService) and shifting 100% of traffic by updating a configuration object via an API.
- **Decision**: We chose **Option 2: Load Balancer Config Update & Reload**.
- **Rationale**:
  - **Predictable and Immediate**: A configuration reload typically takes milliseconds and affects all new connections instantly, unlike DNS which has TTL-based delays and inconsistent client caching.
  - **Connection Draining**: Tools like Nginx and HAProxy support graceful reloads (`nginx -s reload`, `haproxy -sf`), which allow old worker processes to finish serving existing requests before terminating, guaranteeing zero dropped in-flight connections.
  - **Simplicity and Ubiquity**: Reverse proxies are a well-understood, battle-tested technology available in all environments. They don't introduce the complexity of a full service mesh.
  - **Easy Health Check Integration**: The load balancer can be configured to perform active health checks against backends, automatically removing unhealthy nodes. We can leverage this or, at a minimum, integrate our own health check script before triggering the reload.
- **Consequences**:
  - We introduce a single point of failure (the load balancer itself). This is mitigated by using a highly available load balancer pair (e.g., with keepalived).
  - The switching logic is now tied to managing a configuration file and a process signal. This is straightforward to script and automate.
  - We forgo the more granular traffic control (e.g., canary percentages) offered by a service mesh, though this can be added later as an extension.

The following table summarizes the trade-off analysis:

Option	Pros	Cons	Why Not Chosen?
DNS Updates	Simple, no extra infrastructure, works at global scale.	Slow (TTL delays), non-atomic (client caching leads to traffic splitting), impossible to drain connections.	Unacceptable for a requirement of atomic, instant switching with zero dropped connections.
Load Balancer Config Reload	Atomic for new connections, supports connection draining, fast, simple, integrates with health checks.	Single point of failure (SPOF) for the LB itself, requires LB management.	**CHOSEN**. The SPOF is a manageable trade-off, and the benefits align perfectly with our core requirements.
Service Mesh	Extremely fine-grained traffic control, powerful observability, platform-agnostic.	High complexity, significant operational overhead, introduces new failure modes and learning curve.	Overkill for the initial requirement of simple blue-green switching. A good candidate for a future extension.

Common Pitfalls

⚠️ Pitfall: Using `restart` instead of `reload`

- **Description**: Issuing a full restart command (e.g., `systemctl restart nginx`) instead of a graceful reload (`nginx -s reload` or `systemctl reload nginx`). A restart forcibly terminates all worker processes, immediately dropping every in-flight HTTP request and TCP connection, causing user-facing errors and violating the zero-downtime guarantee.
- **Why it's wrong**: It defeats the primary purpose of blue-green deployment. The switch becomes a disruptive event.
- **How to avoid**: Always use the designated graceful reload command for your load balancer. In scripts, explicitly call `nginx -s reload` or the equivalent for HAProxy.

⚠️ Pitfall: Skipping Configuration Validation

- **Description**: Generating a new load balancer configuration file and issuing a reload without first validating its syntax. A single typo in the config can cause the reload to fail, leaving the load balancer in a broken state or, worse, causing it to crash and take the entire service offline.
- **Why it's wrong**: A broken load balancer is a total service outage. The deployment automation must be robust and prevent self-inflicted outages.
- **How to avoid**: Always run the load balancer's built-in config test command (e.g., `nginx -t` or `haproxy -c -f /etc/haproxy/haproxy.cfg`) before triggering a reload. The `switch_traffic` function must incorporate this step and fail fast if validation fails.

⚠️ **Pitfall: Not Performing a Health Check Before Switching**

- **Description**: Switching traffic to the new environment based solely on the fact that its application process is running, without verifying that it can actually serve requests correctly (e.g., database connections, dependent services, internal readiness).
- **Why it's wrong**: You risk redirecting all users to a broken environment. The new version might have a silent bug that only manifests under specific conditions not covered by unit tests.
- **How to avoid**: Integrate the `check_health(env_color)` function from the Environment Manager as a mandatory pre-condition in the `switch_traffic` sequence. The health check should be a deep, application-level check (e.g., hitting the `/health` endpoint and verifying database connectivity). Never switch unless it returns `true`.

⚠️ **Pitfall: Forgetting to Drain Connections from the Old Environment**

- **Description**: While a graceful reload handles this for most reverse proxies, a custom implementation or misuse of APIs might cut off connections to the old environment before long-running requests (e.g., file uploads, WebSocket connections) have completed.
- **Why it's wrong**: Users experience interrupted operations. For stateful connections, this can corrupt user sessions or data.
- **How to avoid**: Rely on the load balancer's built-in graceful termination. Understand how it works: after reload, the old master process signals old workers to finish their current requests and then exit. For advanced scenarios, implement an explicit "drain" mode in the application, where the environment stops accepting new connections but finishes existing ones before the load balancer health check starts failing.

⚠️ **Pitfall: Hard-Coding Environment Addresses**

- **Description**: Writing the IP addresses or hostnames of the blue and green environments directly into the load balancer configuration template. This makes the infrastructure brittle and hard to change.
- **Why it's wrong**: If the environment's IP changes (e.g., after a restart in a cloud environment), the load balancer config is stale and points to a dead server.
- **How to avoid**: Use service discovery mechanisms. In a simple setup, use static hostnames defined in `/etc/hosts` or internal DNS (e.g., `blue-app.internal`, `green-app.internal`). In more dynamic environments, integrate with a discovery service or use upstream groups that can be updated via an API. At a minimum, make the addresses configurable via environment variables or a separate config file.

Implementation Guidance

This section provides concrete starter code and structure for implementing the Load Balancer Controller in Bash, aligned with the primary language requirement.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Load Balancer Software	Nginx (wide adoption, simple config)	HAProxy (more advanced LB features) or Envoy (dynamic configuration via APIs)
Configuration Management	Bash scripts with `sed` / `envsubst` for template replacement	Ansible/Chef/Puppet for idempotent config management, or Consul Template for dynamic updates
Health Check Integration	Custom Bash script calling `curl` on the app's `/health` endpoint	Integrate with the load balancer's native active health checks (e.g., Nginx `health_check` module)

B. Recommended File/Module Structure

Place the Load Balancer Controller scripts in a dedicated directory for clarity and separation of concerns.

```
blue-green-deploy/
├── bin/
│   ├── deploy.sh                # Deployment Orchestrator (main script)
│   └── load-balancer/
│       ├── generate-config.sh   # Load Balancer Controller: Config generator
│       ├── switch-traffic.sh    # Load Balancer Controller: Main switch function
│       └── validate-config.sh   # Load Balancer Controller: Syntax validator
├── config/
│   └── nginx.conf.template      # Jinja2/Envsubst template for Nginx config
├── app/                         # Your application code
└── README.md
```

C. Infrastructure Starter Code

First, here is a complete, reusable Nginx configuration template. This file should be placed at `config/nginx.conf.template`. It uses placeholders that the controller will replace.

```bash
# config/nginx.conf.template

# This is a minimal Nginx config for blue-green routing.

# Variables in ${...} will be substituted by generate-config.sh.


events {

    worker_connections 1024;

}


http {

    # Define upstream blocks for blue and green environments.

    # These point to the actual servers/hosts.

    upstream blue_backend {

        server ${BLUE_HOST}:${BLUE_PORT};

        # Optional: Add load balancing parameters, e.g., 'max_fails=3 fail_timeout=30s'

    }


    upstream green_backend {

        server ${GREEN_HOST}:${GREEN_PORT};

    }


    # A simple health check endpoint for the load balancer itself (optional).

    server {

        listen 8082;

        location /lb-health {

            return 200 "Load Balancer OK\n";

        }

    }


    # Main server block serving actual user traffic.

    server {

        listen 80;

        server_name _;


        # The 'active_backend' variable is set by the generate-config.sh script.

        # It will be replaced with either 'blue_backend' or 'green_backend'.

        location / {

            proxy_pass http://${ACTIVE_BACKEND};

            proxy_set_header Host $host;

            proxy_set_header X-Real-IP $remote_addr;

            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            proxy_set_header X-Forwarded-Proto $scheme;

        }


        # Proxy health checks to the active backend as well.
```

```
        location /health {
            proxy_pass http://${ACTIVE_BACKEND}/health;
        }
    }
}
```

D. Core Logic Skeleton Code

Below are the skeleton scripts for the core Load Balancer Controller functions. The learner must fill in the `TODO` steps.

1. Configuration Generator (`bin/load-balancer/generate-config.sh`):

```bash
#!/usr/bin/env bash

# generate-config.sh

# Generates the final Nginx configuration from a template.

# Usage: ./generate-config.sh <active_environment>

#   active_environment: "blue" or "green"


set -euo pipefail


# --- Configuration ---

readonly TEMPLATE_FILE="../config/nginx.conf.template"

readonly OUTPUT_FILE="/etc/nginx/nginx.conf"

readonly BLUE_PORT="8080"

readonly GREEN_PORT="8081"

# In a real scenario, these hosts might be fetched from service discovery.

readonly BLUE_HOST="localhost"

readonly GREEN_HOST="localhost"

# --- End Configuration ---


if [[ $# -ne 1 ]]; then

    echo "Usage: $0 <active_environment>"

    echo "  active_environment must be 'blue' or 'green'"

    exit 1

fi


ACTIVE_ENV="$1"


case "$ACTIVE_ENV" in

    "blue")

        ACTIVE_BACKEND="blue_backend"

        ;;

    "green")

        ACTIVE_BACKEND="green_backend"

        ;;

    *)

        echo "Error: active_environment must be 'blue' or 'green'. Got: $ACTIVE_ENV"

        exit 1

        ;;

esac


echo "Generating Nginx config with active backend: $ACTIVE_BACKEND ($ACTIVE_ENV)"


# TODO 1: Check if the template file exists. If not, exit with an error.

# TODO 2: Use `envsubst` to substitute environment variables in the template.

#   Hint: Export the variables (ACTIVE_BACKEND, BLUE_HOST, BLUE_PORT, GREEN_HOST, GREEN_PORT) so envsubst can see them.
```

```
#   Command: envsubst '${ACTIVE_BACKEND},${BLUE_HOST},${BLUE_PORT},${GREEN_HOST},${GREEN_PORT}' < "$TEMPLATE_FILE" > "$OUTPUT_FILE"

# TODO 3: Log the action to the deployment log: `echo "$(date): Generated config for $ACTIVE_ENV" >> $DEPLOYMENT_LOG`

# TODO 4: Print a success message.


echo "Configuration generated at $OUTPUT_FILE"
```

2. Configuration Validator (`bin/load-balancer/validate-config.sh`):

```bash
#!/usr/bin/env bash

# validate-config.sh

# Validates the generated Nginx configuration for syntax errors.

# Exits with code 0 if valid, non-zero if invalid.


set -euo pipefail


readonly CONFIG_FILE="/etc/nginx/nginx.conf"


echo "Validating Nginx configuration at $CONFIG_FILE..."


# TODO 1: Check if nginx binary is available. If not, exit with error.

# TODO 2: Run `nginx -t -c $CONFIG_FILE` to test the configuration.

#   This command will output errors and exit with a non-zero code if invalid.

# TODO 3: If validation succeeds, log it and exit 0.

# TODO 4: If validation fails, capture the error output, log it critically, and exit with the nginx command's exit code.


echo "Configuration validation passed."
```

3. Traffic Switch Script (`bin/load-balancer/switch-traffic.sh`):

```bash
#!/usr/bin/env bash

# switch-traffic.sh

# The main function that orchestrates the atomic traffic switch.

# Usage: ./switch-traffic.sh <target_environment>

#   target_environment: "blue" or "green"


set -euo pipefail


# --- Dependencies (paths to other scripts) ---

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

GENERATE_CONFIG_SCRIPT="$SCRIPT_DIR/generate-config.sh"

VALIDATE_CONFIG_SCRIPT="$SCRIPT_DIR/validate-config.sh"

# Assume a health check script exists in a sibling directory.

HEALTH_CHECK_SCRIPT="$SCRIPT_DIR/../environment/health-check.sh"


# --- State File ---

readonly ACTIVE_ENV_FILE="/tmp/active_environment"


# Helper: Get current active environment from state file.

get_current_active_env() {

    if [[ -f "$ACTIVE_ENV_FILE" ]]; then

        cat "$ACTIVE_ENV_FILE"

    else

        # Default to blue if state file doesn't exist (initial state).

        echo "blue"

    fi

}


# Main switch function

switch_traffic() {

    local target_env="$1"

    local current_env

    current_env=$(get_current_active_env)


    echo "Current active environment is: $current_env"

    echo "Requested switch to: $target_env"


    if [[ "$current_env" == "$target_env" ]]; then

        echo "Target environment is already active. No action needed."

        return 0

    fi


    # TODO 1: Perform a deep health check on the target environment.

    #   Call the health check script: `"$HEALTH_CHECK_SCRIPT" "$target_env"`
```

```
    #   If it fails (non-zero exit), abort the switch with an error message.
    #   Example: if ! "$HEALTH_CHECK_SCRIPT" "$target_env"; then echo "Health check failed for $target_env"; exit 1; fi

    # TODO 2: Generate the new load balancer configuration for the target environment.
    #   Call: "$GENERATE_CONFIG_SCRIPT" "$target_env"

    # TODO 3: Validate the newly generated configuration.
    #   Call: "$VALIDATE_CONFIG_SCRIPT"
    #   If validation fails, the script will exit, aborting the switch.

    # TODO 4: Gracefully reload the load balancer to apply the new config.
    #   Command: `nginx -s reload`
    #   Hint: Check the exit code of this command. If it fails, log an error and consider exiting.

    # TODO 5: Update the persistent state file to reflect the new active environment.
    #   Command: `echo "$target_env" > "$ACTIVE_ENV_FILE"`

    # TODO 6: Log the successful switch to the deployment log.
    #   Command: `echo "$(date): Traffic switched from $current_env to $target_env" >> $DEPLOYMENT_LOG`

    echo "Traffic switch completed successfully. Active environment is now: $target_env"
}

# Script entry point
if [[ $# -ne 1 ]]; then
    echo "Usage: $0 <target_environment>"
    echo "  target_environment must be 'blue' or 'green'"
    exit 1
fi

target="$1"
switch_traffic "$target"
```

E. Language-Specific Hints (Bash)

- **Error Handling**: Always use `set -euo pipefail` at the top of scripts. This makes the script exit on any error (`-e`), undefined variable usage (`-u`), and pipe failures (`pipefail`).
- **Configuration Templating**: The `envsubst` command (from the `gettext` package) is perfect for simple variable substitution in templates. Install it via `apt-get install gettext-base` or `yum install gettext`.
- **Graceful Reload**: For Nginx, always use `nginx -s reload`. For HAProxy, use `haproxy -f /etc/haproxy/haproxy.cfg -p /var/run/haproxy.pid -sf $(cat /var/run/haproxy.pid)` to smoothly restart.
- **State Persistence**: Writing to `/tmp/active_environment` is simple but volatile (may not survive a system reboot). For production, use a more persistent location like `/var/lib/blue-green/state` or a key-value store like Redis/Consul.

F. Milestone Checkpoint

After implementing the Load Balancer Controller, you should be able to manually test traffic switching.

1. **Start both environments** (using the Environment Manager from Milestone 1):

```bash
./bin/environment/start-environment.sh blue

./bin/environment/start-environment.sh green
```

2. **Initial State**: Ensure the load balancer is pointing to blue (the default). You can check by querying the load balancer's main endpoint:

```bash
curl http://localhost:80/
```

`The response should include `"environment": "blue"`.`

3. **Test the Switch**:

```bash
sudo ./bin/load-balancer/switch-traffic.sh green
```

`Observe the logs. It should generate config, validate it, and reload Nginx.`

4. **Verify the Switch**:

```bash
curl http://localhost:80/
```

`The response should now show `"environment": "green"`. Also, check the state file:`

```bash
cat /tmp/active_environment
```

`It should contain `green`.`

5. **Test Rollback**:

```bash
sudo ./bin/load-balancer/switch-traffic.sh blue
```

`Traffic should instantly revert to blue.`

Expected Output Signs of Success:

- The `switch-traffic.sh` script runs without errors.
- The `curl` command returns the correct environment color immediately after the switch.
- The Nginx error log (`/var/log/nginx/error.log`) shows no critical errors during reload.
- No "502 Bad Gateway" or connection errors are observed during the switch.

Signs of Trouble:

- `curl` fails with "Connection refused": The load balancer may have crashed due to a bad config. Check Nginx status (`systemctl status nginx`) and the validation step logs.
- `curl` still shows the old environment: The reload may have failed silently. Check if the config file was actually updated and if the Nginx master process received the signal. Use `ps aux | grep nginx` to see if old workers are still running.
- Health check fails: The target environment's `/health` endpoint may be unhealthy. Verify the environment is running and its dependencies (database) are connected.

> **Milestone(s):** Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

Component: Deployment Orchestrator

The **Deployment Orchestrator** is the central automation engine that sequences, coordinates, and executes the entire blue-green deployment process from code commit to traffic switch. Think of it as the conductor of a symphony, ensuring each musician (component) plays their part at the right time to create a harmonious release without interrupting the audience's experience. While the `Environment Manager` builds the stages and the `Load Balancer Controller` directs the audience, the orchestrator manages the entire show's schedule, cues the scene changes, and has a contingency plan if a musician misses a note. Its primary responsibility is to replace error-prone, manual deployment checklists with a reliable, repeatable, and auditable automated sequence that guarantees zero-downtime and enables instant rollback.

Mental Model: Conductor of an Orchestra

Imagine a symphony orchestra performing a continuous concert that must never stop. The music represents your live user traffic and service availability. The orchestra has two identical sections: the **Blue Section** and the **Green Section**, each capable of playing the entire symphony. Only one section performs for the audience at a time. You need to introduce a revised piece of music (a new software version).

The **Conductor** (Deployment Orchestrator) manages this transition:

1. **Score Study (Build):** First, the conductor receives the new musical score (the new application version) and verifies it's correctly formatted.
2. **Section Rehearsal (Deploy to Inactive):** While the Blue Section performs live, the conductor has the Green Section silently rehearse the new score in a separate room. They don't interrupt the live performance.
3. **Sound Check (Smoke Tests):** Before letting the new section perform, the conductor quickly tests them—do the instruments work? Can they play the key passages? This is a critical, fast validation.
4. **Cue the Switch (Traffic Switch):** Once satisfied, the conductor gives a subtle signal. The stage manager (Load Balancer Controller) seamlessly redirects the audience's attention from the Blue Section to the Green Section. The music never stops.
5. **Contingency Plan (Rollback):** If during the sound check the Green Section falters, the conductor immediately abandons the switch. The Blue Section continues playing, and the audience is none the wiser.

This model highlights the orchestrator's role: **coordination, timing, validation, and fallback.** It doesn't play the instruments (run the application) or move the audience (route traffic), but it tells the specialized components *when* and *how* to do their jobs to achieve the overall goal of a flawless, uninterrupted performance.

Interface and Responsibilities

The Deployment Orchestrator is typically implemented as a script or a set of scripts with a well-defined entry point (e.g., `deploy.sh v2.1.0`). Its interface consists of functions that call out to other components and logical steps that form the deployment pipeline. The core responsibility is to execute the **blue-green deployment algorithm** idempotently, meaning running it multiple times with the same inputs yields the same safe outcome without causing errors or double-deployments.

The primary function, often invoked from a CI/CD pipeline trigger, follows this high-level sequence:

Main Deployment Algorithm:

1. **Determine State:** Identify which environment (blue or green) is currently active and which is inactive.
2. **Build Artifact:** Compile, package, or containerize the new version of the application. This step is idempotent—building the same commit twice should produce an identical artifact.
3. **Deploy to Inactive:** Deploy the new artifact *only* to the environment that is not receiving live traffic. This is the core isolation benefit of blue-green.
4. **Validate Deployment:** Perform health checks and a suite of smoke tests against the newly deployed, inactive environment to ensure basic functionality works.
5. **Execute Traffic Switch:** If validation passes, atomically reconfigure the load balancer to direct all new traffic to the newly deployed environment, which becomes the active one.
6. **Verify Post-Switch:** After the switch, run a second set of quick checks against the *newly active* environment to confirm it handles live traffic correctly.
7. **Cleanup (Optional):** Optionally, stop the old, now-inactive environment to free resources, or leave it running for instant rollback capacity.

The orchestrator also provides a **rollback function** that can be triggered manually or automatically if the validation step fails. This function's logic is simple: switch traffic back to the previously active environment.

The table below defines the key functions that comprise the orchestrator's interface. These functions often wrap or call the interfaces of the `Environment Manager` and `Load Balancer Controller`.

Method Name	Parameters	Returns	Description
`deploy_to_environment(env_color, version)`	`env_color` (string): "blue" or "green" `version` (string): The version tag or commit hash to deploy.	`void`	Orchestrates the deployment of a specific version to a target environment. This involves calling `stop_environment`, updating the environment's configuration, calling `start_environment`, and waiting for `check_health` to pass. It is a higher-level function that uses the `Environment Manager`.
`run_smoke_tests(env_color, port)`	`env_color` (string): "blue" or "green" `port` (integer): The port number the environment is listening on (e.g., `BLUE_PORT` or `GREEN_PORT`).	`boolean` (`true` for success, `false` for failure)	Validates the basic functionality of a deployed environment by making HTTP requests to critical endpoints (e.g., `/health`, `/api/v1/status`). It verifies responses match expected status codes, contain required fields like `environment` and `version`, and perform any essential business logic checks.
`switch_traffic(target_env)`	`target_env` (string): The environment ("blue" or "green") to make active.	`void`	Atomically updates the load balancer configuration to route all traffic to the `target_env` after validating it is healthy via `check_health`. This function should call the Load Balancer Controller's configuration generation and graceful reload functions.
`rollback_deployment(failed_env)`	`failed_env` (string): The environment that just failed deployment or validation.	`void`	Reverts traffic to the *other* environment (the one that was active before the failed deployment attempt). This is essentially a specialized call to `switch_traffic` with the previous environment as the target. It may also trigger alerts or log the failure.
`get_inactive_environment()`	None	`string` ("blue" or "green")	Determines which environment is not currently receiving live traffic by reading the persistent state (e.g., `ACTIVE_ENV_FILE`). If blue is active, returns "green", and vice-versa.
`get_current_active_env()`	None	`string`	Reads the currently active environment from the persistent state store (e.g., `ACTIVE_ENV_FILE`). This is the source of truth for the system's routing state.

State Tracking: The orchestrator relies on a simple, persistent state to know which environment is active. This is often a file (e.g., `ACTIVE_ENV_FILE = /tmp/active_environment`) containing either the string "blue" or "green". The `Load Balancer Controller` writes to this file after a successful switch, and the orchestrator reads from it to make decisions. This shared state is critical for consistency.

Architecture Decision: Deployment Automation Style

Decision: Imperative Scripting for Orchestration

- **Context:** We need a lightweight, understandable, and debuggable automation layer that integrates with existing CI/CD tools and can run on any agent (bare metal, VM, container). The team has strong Bash/Python skills, and the deployment process is a linear sequence of steps with clear failure points.
- **Options Considered:**
  1. **Imperative Scripting (Bash/Python):** Write a main deployment script that executes commands in a specific order, with explicit error handling and logging at each step.
  2. **Declarative State-Based (Terraform, Ansible, Kubernetes):** Define the desired end-state (e.g., "version 2.1 is active") and let a tool reconcile the current state to match, potentially with more complex operators or custom resources.
  3. **Pipeline-as-Code (Jenkinsfile, GitLab CI, GitHub Actions):** Define the deployment sequence within the CI/CD platform's native pipeline definition language.
- **Decision:** We chose **Imperative Scripting (Bash)** as the core of the Deployment Orchestrator.
- **Rationale:**
  - **Transparency & Debuggability:** The exact sequence of operations is visible in the script. When a deployment fails, engineers can follow the script line-by-line, add debug output, or run commands manually. There is no "magic" reconciliation loop to understand.
  - **Simplicity & Portability:** A Bash script has minimal dependencies (just a shell) and can run anywhere. It's easier for team members of all levels to read and modify.
  - **Explicit Control:** Blue-green deployments require precise timing and conditional logic (e.g., "if smoke tests pass, then switch; else, rollback"). Imperative scripts excel at expressing this procedural logic clearly.
  - **Integration:** It can easily be invoked from *any* CI/CD system (by simply calling the script), making our deployment process tool-agnostic.
- **Consequences:**
  - **We must write robust error handling.** Imperative scripts fail where they break; we need to catch errors, log them, and trigger rollbacks explicitly.
  - **State management is our responsibility.** The script must read/write the `ACTIVE_ENV_FILE` and manage its own idempotency.
  - **It may become complex over time.** As we add more steps (database migrations, notifications, complex rollbacks), the main script can grow. This can be mitigated by breaking it into modular functions and files.

Option	Pros	Cons	Chosen?
Imperative Scripting (Bash)	Simple, portable, transparent, easy to debug, explicit control flow.	Manual error handling, script can become monolithic, state management is manual.	**Yes**
Declarative State-Based	Idempotent by design, can handle complex infrastructure drift, strong community tools.	Higher learning curve, "black box" reconciliation can be hard to debug, overkill for linear deployment sequence.	No
Pipeline-as-Code	Tight integration with CI/CD platform, native logging and retry mechanisms.	Vendor lock-in, less portable, may be less expressive for complex logic than a full scripting language.	No (but used as the trigger)

Common Pitfalls

Learners implementing the Deployment Orchestrator often stumble on these specific, concrete mistakes:

⚠️ Pitfall 1: Missing Error Handling After Each Step

- **Description:** Writing a script that assumes every command succeeds (e.g., `deploy_to_environment "green" "v2.1" && run_smoke_tests "green" $GREEN_PORT && switch_traffic "green"`). If `deploy_to_environment` fails, the script might still attempt to run smoke tests on a non-existent service, causing confusing errors.
- **Why it's wrong:** The deployment process loses its reliability. A failure in an early step can cascade, causing undefined behavior, wasted time debugging, and potentially leaving the system in an inconsistent state (e.g., a partially deployed environment).
- **Fix:** Implement explicit error checking after *every* operation that can fail. Use Bash's `set -euo pipefail` at the top of the script, and/or wrap logical sections in `if ! ...; then` blocks to catch failures and immediately jump to a rollback or cleanup routine.

⚠️ Pitfall 2: No Timeouts for Health Checks and Smoke Tests

- **Description:** The script calls `check_health` or `run_smoke_tests`, which sends an HTTP request but does not set a timeout. If the application is stuck in a crash loop or is extremely slow, the request could hang indefinitely, causing the deployment pipeline to stall forever.
- **Why it's wrong:** This creates a "zombie" deployment that consumes CI/CD resources and requires manual intervention to kill. It defeats the purpose of automation.
- **Fix:** Always use timeouts for network calls. In Bash, use `curl --max-time 10` or `timeout 15 curl ...`. In the orchestrator functions, implement retry logic with a maximum total wait period (e.g., "retry health check 6 times with 5-second delays, fail after 30 seconds").

⚠️ Pitfall 3: Not Verifying the Switch Outcome

- **Description:** The script calls `switch_traffic` and immediately exits with success, assuming the load balancer reconfiguration worked. However, the graceful reload could have failed due to a syntax error in the generated config, or the new environment might have a latent bug only under load.
- **Why it's wrong:** You have a false sense of security. Users might be hitting errors, or traffic might still be going to the old version. The deployment is not truly verified.
- **Fix:** After `switch_traffic`, add a post-switch verification step. This can be a quick health check against the load balancer's public endpoint to confirm it returns responses from the *new* environment (check the `environment` field in the response). Log this verification explicitly.

⚠️ Pitfall 4: Hard-Coding Paths and Ports in the Main Script

- **Description:** Scattering values like `BLUE_PORT=8080` and `GREEN_PORT=8081` throughout the script logic, or using absolute paths like `/opt/app/config.sh`.
- **Why it's wrong:** It makes the script inflexible and hard to test in different environments (e.g., staging vs. production). Changing a port requires hunting through the code.
- **Fix:** Source all configuration from a central location at the start of the script. Use environment variables or a dedicated configuration file (e.g., `source ./deploy.config`). Use the provided constant names (`BLUE_PORT`, `GREEN_PORT`) which should be defined in that config.

⚠️ Pitfall 5: Inadequate Logging

- **Description:** The script runs but only outputs `"Deployment started"` and `"Deployment finished"`. When it fails, there's no record of what step failed, what the error message was, or what the system state was.
- **Why it's wrong:** Debugging requires re-running the script with extra `echo` statements or guessing. In a CI/CD environment, logs are the primary source of truth for what happened.
- **Fix:** Implement structured logging. Before each major step, log a `INFO` message with a timestamp (e.g., `log_info "Deploying version $VERSION to $TARGET_ENV"`). On success, log `SUCCESS`; on failure, log `ERROR` with the specific command output. Ensure all logs are written to both stdout and a file (`DEPLOYMENT_LOG`).

Implementation Guidance

This section provides the concrete Bash scaffolding to build your Deployment Orchestrator.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Orchestration Core	Bash script with functions (`deploy.sh`)	Python script with classes and logging (`orchestrator.py`)
HTTP Client for Tests	`curl` with `jq` for JSON parsing	Dedicated test suite using `pytest` / `requests` (Python) or `go test` (Go)
State Storage	Plain text file (`ACTIVE_ENV_FILE`)	Key-Value store (Redis) or database table for audit history
Configuration	Sourced Bash environment file (`.env`)	Configuration management (HashiCorp Vault, AWS Parameter Store)

B. Recommended File/Module Structure:

```
blue-green-project/
├── scripts/
│   ├── orchestrator.sh              # Main deployment orchestrator (THIS COMPONENT)
│   ├── environment_manager.sh       # Environment Manager component
│   ├── load_balancer_controller.sh  # Load Balancer Controller component
│   └── database_migrator.py         # Database Migrator component
├── app/
│   └── (your application code)
├── config/
│   ├── blue.env                     # Blue environment configuration
│   ├── green.env                    # Green environment configuration
│   └── loadbalancer/
│       └── nginx.conf.template      # Nginx configuration template
├── state/
│   └── active_environment           # Persistent state file (created by scripts)
├── logs/
│   └── deployments.log              # Deployment log (append-only)
└── README.md
```

C. Infrastructure Starter Code (Logging Helper): A robust logging function is a prerequisite for good orchestration. Place this at the top of your `orchestrator.sh`.

```bash
#!/usr/bin/env bash

set -euo pipefail  # Strict error handling


# Deployment Orchestrator for Blue-Green Deployments

# Usage: ./orchestrator.sh <version_tag>


# --- Configuration (source from external file if possible) ---

BLUE_PORT=8080

GREEN_PORT=8081

ACTIVE_ENV_FILE="./state/active_environment"

DEPLOYMENT_LOG="./logs/deployments.log"

HEALTH_CHECK_TIMEOUT=5  # seconds

SMOKE_TEST_TIMEOUT=10   # seconds


# --- Logging Utilities ---

log_message() {

    local level=$1

    local message=$2

    local timestamp=$(date -u '+%Y-%m-%d %H:%M:%S UTC')

    echo "[$timestamp] [$level] $message" | tee -a "$DEPLOYMENT_LOG"

}


log_info() {

    log_message "INFO" "$1"

}


log_error() {

    log_message "ERROR" "$1"

}


log_success() {

    log_message "SUCCESS" "$1"

}


# Fatal error: log and exit

fatal_error() {

    log_error "$1"

    exit 1

}


# --- Core Orchestrator Functions (TO BE IMPLEMENTED) ---

# The functions below are skeletons. Implement them according to the algorithm.
```

D. Core Logic Skeleton Code: Here are the key function skeletons with detailed TODOs that map directly to the algorithm steps described in prose.

BASH

```bash
# Main deployment function. Called with a version tag.
deploy_new_version() {
    local version=$1
    log_info "Starting deployment of version: $version"

    # TODO 1: Determine the currently active environment by reading ACTIVE_ENV_FILE.
    #   - If file doesn't exist, default to "blue" and create the file.
    #   - Store result in variable 'current_active'.

    # TODO 2: Based on 'current_active', determine the 'target_env' (inactive environment).
    #   - If current_active is "blue", target_env="green". Else, target_env="blue".
    #   - Log the decision.

    # TODO 3: Build the application artifact for $version.
    #   - This could be `docker build -t myapp:$version .` or compiling code.
    #   - Check the build command's exit status. On failure, call fatal_error.

    # TODO 4: Deploy the $version to the $target_env.
    #   - Call a helper function `deploy_to_environment "$target_env" "$version"`.
    #   - This function should stop the env, update it, start it, and wait for health.
    #   - Implement a retry loop for health checks with a timeout.

    # TODO 5: Run smoke tests against the newly deployed $target_env.
    #   - Determine the correct port (BLUE_PORT or GREEN_PORT) for $target_env.
    #   - Call `run_smoke_tests "$target_env" "$port"`.
    #   - If tests fail, call `trigger_rollback "$target_env" "$current_active"` and exit.

    # TODO 6: If smoke tests pass, execute the traffic switch.
    #   - Call `switch_traffic "$target_env"`.
    #   - Verify the switch was successful by checking the active environment file updated.

    # TODO 7: Perform post-switch verification.
    #   - Wait a few seconds for load balancer propagation.
    #   - Send a health check to the public endpoint (or load balancer) and confirm the
    #     response contains environment=$target_env and version=$version.

    log_success "Deployment of $version to $target_env completed successfully."
}

# Deploys a specific version to a specific environment (blue or green).
deploy_to_environment() {
    local env_color=$1
    local version=$2
    log_info "Deploying version $version to $env_color environment."
```

```
    # TODO 1: Determine the environment-specific configuration.
    #   - Source the config file for this env (e.g., "./config/${env_color}.env").

    # TODO 2: Stop the environment if it is running.
    #   - Call `stop_environment` from environment_manager.sh, or send a stop signal.

    # TODO 3: Update the environment with the new version.
    #   - This could involve copying new binaries, updating a Docker image tag in a
    #     docker-compose file, or deploying a new Kubernetes manifest.
    #   - Ensure all environment variables (like APP_VERSION) are set correctly.

    # TODO 4: Start the environment.
    #   - Call `start_environment` from environment_manager.sh.

    # TODO 5: Wait for the environment to become healthy.
    #   - Implement a retry loop (e.g., 12 attempts with 5 second delay).
    #   - In each attempt, call `check_health` for the environment.
    #   - If health check succeeds, break and log success.
    #   - If all attempts fail, log error and return a failure status (use `return 1`).
}

# Runs a suite of basic HTTP tests to validate the environment.
run_smoke_tests() {
    local env_color=$1
    local port=$2
    log_info "Running smoke tests for $env_color on port $port."

    local base_url="http://localhost:$port"

    # TODO 1: Test the health endpoint.
    #   - Use `curl --max-time $SMOKE_TEST_TIMEOUT -s -f $base_url/health`
    #   - Check exit code. If non-zero, test fails.
    #   - Optionally, parse JSON response and verify `status` field is "healthy".

    # TODO 2: Test a critical business API endpoint (if your app has one).
    #   - Example: `curl --max-time $SMOKE_TEST_TIMEOUT -s $base_url/api/v1/status`
    #   - Verify HTTP status code is 200.
    #   - Use `jq` to check the response contains expected fields like `environment` and `version`.

    # TODO 3: Add any other quick, non-destructive checks that prove the app works.
    #   - For a web app: fetch the homepage and check for a key string.
    #   - For an API: validate the response schema.

    # TODO 4: If any test fails, log the specific failure and return 1 (false).
    #   If all pass, log success and return 0 (true).
```

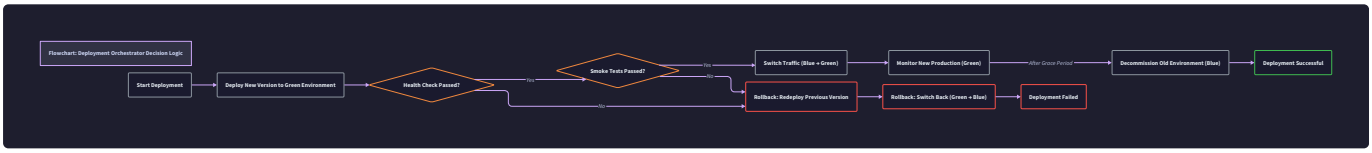
```
}

# Switches traffic by updating the load balancer configuration.
switch_traffic() {
    local target_env=$1
    log_info "Initiating traffic switch to $target_env."

    # TODO 1: Validate the target environment is healthy.
    #   - Call `check_health` for $target_env. If unhealthy, log error and exit.

    # TODO 2: Generate the new load balancer configuration.
    #   - Call `generate_config_for_env "$target_env"` from load_balancer_controller.sh.

    # TODO 3: Validate the generated configuration file syntax.
    #   - For nginx: `nginx -t -c <generated_config_path>`.
    #   - If validation fails, log error and exit.

    # TODO 4: Atomically apply the new configuration with a graceful reload.
    #   - For nginx: `nginx -s reload`.
    #   - Ensure the reload command does not drop connections (graceful).

    # TODO 5: Update the persistent state file.
    #   - Write "$target_env" to the `ACTIVE_ENV_FILE`.
    #   - Log the state change.

    log_success "Traffic switched to $target_env."
}

# Rolls back by switching traffic to the previous environment.
trigger_rollback() {
    local failed_env=$1
    local previous_env=$2  # The environment that was active before this deployment
    log_error "Deployment to $failed_env failed. Initiating rollback to $previous_env."

    # TODO 1: Call `switch_traffic "$previous_env"`.
    # TODO 2: Send alert/notification about the rollback (e.g., echo to slack webhook).
    # TODO 3: Optionally, stop the failed environment to conserve resources.
}

# Helper to get the currently inactive environment.
get_inactive_environment() {
    # TODO: Read ACTIVE_ENV_FILE, if blue return green, if green return blue.
    # Handle file not existing (default to green?).
}
```

E. Language-Specific Hints (Bash):

- **Error Checking:** Always check exit codes with `$?` or use `set -e`. For conditional logic, use `if ! command; then ... fi`.
- **JSON Parsing:** Install `jq` for robust parsing of JSON responses from health checks and APIs (e.g., `curl -s ... | jq -r '.environment'`).
- **Retry Loops:** Implement retries with exponential backoff for health checks. A simple pattern: `for i in {1..6}; do if check_health; then break; fi; sleep 5; done`.
- **Argument Parsing:** Use `getopts` for more complex script arguments, or simply use positional arguments (`$1`, `$2`).

F. Milestone Checkpoint (Milestone 3): After implementing the Deployment Orchestrator, you should be able to run a full deployment without manual intervention.

1. **Command to Run:** `./scripts/orchestrator.sh v2.1.0`
2. **Expected Output:** The script should log each step to both the console and the `DEPLOYMENT_LOG`. It should end with a `[SUCCESS]` message.
3. **Manual Verification:**
   - While the script runs, use `curl http://localhost:8080/health` (or your load balancer port) repeatedly. The response should never be interrupted or return an error.
   - After completion, the health endpoint should report the new version (`v2.1.0`) and the environment color should have switched (e.g., from "blue" to "green").
   - Check the `ACTIVE_ENV_FILE` contains the correct environment.
4. **Signs of Trouble:**
   - **Script hangs:** Likely a health check timeout. Check if the application in the target environment started correctly. Look at its logs.
   - **Traffic didn't switch:** The `ACTIVE_ENV_FILE` may not be updated, or the nginx reload failed. Check the nginx error log (`/var/log/nginx/error.log`).
   - **Rollback triggered:** The smoke tests failed. Examine the smoke test logs and check the application logs in the target environment for errors.



> **Milestone(s):** Milestone 4 (Rollback & Database Migrations)

Component: Database Migrator

The **Database Migrator** is the guardian of data integrity and backward compatibility during deployments. While the Environment Manager handles application instances and the Load Balancer Controller directs traffic, the Migrator ensures the underlying data schema evolves safely alongside the code. In a blue-green deployment, both the old (active) and new (inactive) application versions must run simultaneously for a brief period during the traffic switch. This creates a critical constraint: **both application versions must function correctly with the current database schema**. The Database Migrator's core responsibility is to orchestrate schema changes in a way that maintains this bidirectional compatibility, enabling zero-downtime deployments and instant rollbacks.

Mental Model: Expanding a Bridge While Traffic Flows

Imagine a busy, two-lane bridge that cannot be closed. You need to replace it with a newer, three-lane bridge. The **expand-contract** pattern is the engineering playbook for this scenario:

1. **Expand:** First, you build a new, third lane *alongside* the existing two. Traffic continues to flow uninterrupted on the original lanes. This is analogous to adding a new, nullable column to a database table. The old application (using only lanes 1 and 2) and the new application (which can use lane 3) both work.
2. **Traffic Shift:** Once the new lane is complete and tested, you redirect all traffic to use the new configuration—perhaps now treating lanes 2 and 3 as the main thoroughfare, while lane 1 becomes the shoulder. This is the deployment of the new application code and the subsequent traffic switch. The new code uses the new column (lane 3), but the schema still supports the old code.
3. **Contract:** After confirming the new traffic pattern is stable and the old lane 1 is no longer in active use, you safely demolish the original lane 1 and repave the area. This corresponds to removing the old, now-obsolete column from the database schema.

This model emphasizes that the "expand" (additive) phase must come *before* the code deployment that depends on it, and the "contract" (subtractive) phase must come *after* the old code is completely retired and any necessary data migration is verified. The Migrator's job is to execute these schema changes as idempotent, versioned scripts, ensuring the bridge (database) is always safe for the traffic (application requests) flowing across it.

Interface and Responsibilities

The Database Migrator exposes a minimal interface focused on forward progress and safety. It does not manage application connections but ensures the schema is in a state compatible with the application versions about to run.

Method Name	Parameters	Returns	Description
`apply_migrations`	`target_version` (string), `direction` (string, optional)	`boolean` (success)	Applies all migration scripts up to (or down to) the `target_version`. The `direction` defaults to "up". This function must be **idempotent**; running it multiple times for the same target version should have no effect. It is responsible for transactional safety where supported (e.g., wrapping migrations in a transaction for RDBMS like PostgreSQL).
`get_current_schema_version`	None	`string`	Reads and returns the current version of the database schema from a dedicated version table (e.g., `schema_migrations`). This is used to determine which migrations need to be applied.
`validate_migration_compatibility`	`app_version_blue`, `app_version_green`	`boolean`	**Critical Pre-flight Check.** Validates that the current database schema is compatible with both the blue and green application versions. This might check that required columns exist, have correct data types, and that no "contract" steps have been executed prematurely that would break the older application.

Internally, the component manages:

- **Migration Scripts:** A sequential set of versioned SQL (or other database-specific) files, e.g., `001_add_user_table.sql`, `002_add_email_column.sql`.
- **Version Tracking Table:** A simple table (e.g., `schema_migrations`) with a single column `version` to record which migrations have been successfully applied. This provides idempotency and history.
- **Expand and Contract Phase Markers:** A mechanism to track which phase of a multi-step migration is complete. This could be an additional column in the version table or a separate `migration_phases` table.

Architecture Decision: Database Migration Strategy

> **Decision: Expand-Contract Pattern for Schema Migrations**
>
> - **Context**: The blue-green deployment model requires the database to support two different, concurrently running application versions during the traffic switch. A traditional "break-and-fix" migration that immediately alters or drops columns would cause the older application version to fail, breaking rollback capability and potentially causing downtime.
> - **Options Considered**:
>   1. **Expand-Contract Pattern**: All schema changes are performed in two or more phases. Phase 1 ("expand") adds new structures (tables, nullable columns) without removing old ones. Phase 2 ("contract") removes deprecated structures only after all application code using them has been retired.
>   2. **Versioned APIs/Service Layer**: The application itself is designed to support multiple versions of its data access layer or API endpoints. The database schema can change more freely, as the application contains logic to handle both old and new formats, often translating between them.
>   3. **Dual-Write/Shadow Write Pattern**: During the transition, the application writes data to both the old and new structures simultaneously. This is useful for migrating data between fundamentally different schemas (e.g., changing a column type) but adds significant application complexity.
> - **Decision**: Adopt the **Expand-Contract Pattern** as the primary strategy for database migrations.
> - **Rationale**: This pattern provides the strongest guarantee of backward compatibility with minimal runtime performance overhead. It keeps the complexity contained within the migration scripts and the deployment process, rather than polluting the application logic with version checks and data transformations. It aligns perfectly with the blue-green model: the "expand" is applied before deploying the new code, and the "contract" is applied in a later, separate deployment cycle after the old code is no longer in production. It is also well-understood and supported by many migration tools.
> - **Consequences**: **Enables**: Truly zero-downtime, backward-compatible schema changes. Safe, instant rollback at the application level. **Trade-offs**: Requires disciplined, forward-looking planning of schema changes. The database may temporarily carry some "dead weight" (unused columns or tables) until the contract phase is complete. Some complex data transformations may require additional interim steps.

Option	Pros	Cons	Chosen?
Expand-Contract Pattern	Strong backward compatibility; low runtime overhead; clean separation of concerns.	Requires multi-step migration planning; temporary schema cruft.	**Yes** (Primary strategy)
Versioned APIs	Allows more flexible, independent evolution of services.	High complexity in application code; potential performance cost for translation logic.	No (Overkill for monolithic app)
Dual-Write Pattern	Excellent for complex, multi-stage data backfills and transformations.	Doubles write load; significantly increases application and operational complexity.	No (Use as a tactical tool within expand-contract if needed)

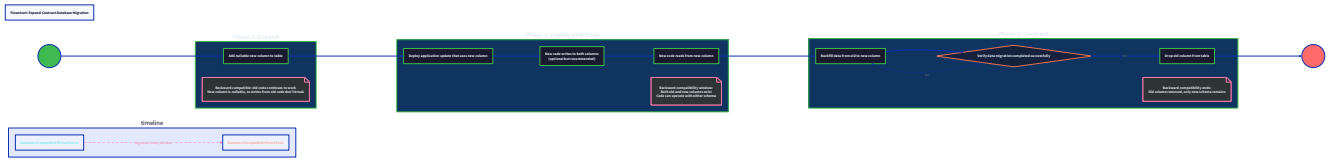
The process for a single schema change, such as renaming a column from `username` to `user_name`, is broken into three distinct deployments:

1. **Deployment A (Expand):**
   - Migration Script: `ALTER TABLE users ADD COLUMN user_name VARCHAR(255);`

- Application Code: **Old version** (uses `username`). New column is ignored.
- State: Both columns exist. Application writes to `username`, reads from `username`. The new `user_name` is `NULL`.

2. **Deployment B (Migrate & Switch):**
   - Migration Script: `UPDATE users SET user_name = username WHERE user_name IS NULL;` (Backfill data). May also add application-level dual-write logic if needed.
   - Application Code: **New version** (uses `user_name`). Code is updated to read from and write to `user_name`.
   - State: Both columns exist and are synchronized. Traffic is switched from old app (using `username`) to new app (using `user_name`). Rollback is still safe because the old `username` column still has valid data.
3. **Deployment C (Contract):**
   - Migration Script: `ALTER TABLE users DROP COLUMN username;`
   - Application Code: **New version** (still uses `user_name`). The old column is no longer referenced.
   - State: Only the `user_name` column remains. The schema is cleaned up.

This phased approach is visualized in the flowchart:



Common Pitfalls

⚠️ Pitfall: Breaking Backward Compatibility with a Single-Step Migration

- **Description:** Writing a migration that immediately drops or renames a column that the currently active (old) application version still requires.
- **Why it's Wrong:** The moment this migration runs, the active environment's application will start throwing fatal database errors (e.g., "column not found"), causing a service outage. The entire purpose of blue-green—zero-downtime—is defeated.
- **Fix:** Always follow the expand-contract pattern. Use `ADD COLUMN` with `NULL` default first. Deploy the new code that uses the new column. *Then*, in a later deployment, `DROP COLUMN` the old one.

⚠️ Pitfall: Applying the "Contract" Phase Too Early

- **Description:** Running the `DROP COLUMN` migration in the same deployment cycle as the code change that starts using the new column.
- **Why it's Wrong:** This destroys the rollback window. If the new deployment fails its smoke tests and traffic is switched back to the old environment, the old code will immediately fail because its required column is gone. Rollback becomes impossible without restoring a database backup, which is slow and complex.
- **Fix:** The contract phase (removing old structures) must be a separate, subsequent deployment. Only after the new version has been live, stable, and the old version is confirmed to be out of use (e.g., after several days or a full deployment cycle) should the cleanup migration be applied.

⚠️ Pitfall: Forgetting to Backfill Data in the Expand Phase

- **Description:** Adding a new `NOT NULL` column without a default value, or adding a new column and assuming the application will populate it for all new records without handling existing ones.
- **Why it's Wrong:** The `ALTER TABLE ... ADD COLUMN ... NOT NULL` will fail on existing rows. Even if it's nullable, the new application logic might crash when reading `NULL` values from old records, assuming the field is always populated.
- **Fix:** For `NOT NULL` columns, either add it as nullable, backfill the data, then apply the `NOT NULL` constraint, or provide a sensible default value. Always write data backfill logic as part of the migration script.

⚠️ Pitfall: Not Testing with Both Application Versions

- **Description:** Testing the new application version with the new schema, but not validating that the old application version still works correctly with the post-expand schema.
- **Why it's Wrong:** The old version might have subtle bugs or incompatibilities, such as a query that does a `SELECT *` and breaks when an unexpected new column is returned, or an ORM that eagerly loads all columns and fails on a new data type.
- **Fix:** As part of the deployment automation, the `validate_migration_compatibility` function should be called. Furthermore, integration tests should spin up instances of *both* application versions against the migrated database and run their respective test suites.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Migration Script Language	Plain SQL files	A dedicated migration tool (e.g., Flyway, Liquibase, Goose)
Version Tracking	A simple `schema_migrations` table managed by your scripts	Leverage the versioning system of an advanced migration tool
Execution Engine	Bash script calling `psql` / `mysql` CLI	Integration into the application startup (e.g., using `golang-migrate`)

B. Recommended File/Module Structure

```
project-root/
├── deployments/
│   └── orchestrate.sh          # Main Deployment Orchestrator (calls the migrator)
├── database/
│   ├── migrations/
│   │   ├── 001_initial_schema.up.sql
│   │   ├── 001_initial_schema.down.sql
│   │   ├── 002_add_user_profile.up.sql # Expand: add nullable `bio` column
│   │   └── 002_add_user_profile.down.sql
│   ├── migrate.sh              # Database Migrator wrapper script
│   └── utils.py                # Python helpers for version tracking & validation
└── apps/
    ├── blue-green-app/         # The application code
    └── ...
```

C. Infrastructure Starter Code

`database/utils.py` – Python helper for idempotent migration tracking:

PYTHON

```python
#!/usr/bin/env python3
"""
Database migration utilities for version tracking and idempotency.
"""
import os
import sys
import argparse
import subprocess
from pathlib import Path

# Configuration - adjust based on your database
DB_HOST = os.getenv('DB_HOST', 'localhost')
DB_PORT = os.getenv('DB_PORT', '5432')
DB_NAME = os.getenv('DB_NAME', 'myapp')
DB_USER = os.getenv('DB_USER', 'myapp_user')
DB_PASSWORD = os.getenv('DB_PASSWORD', '')

MIGRATIONS_DIR = Path(__file__).parent / 'migrations'
VERSION_TABLE = 'schema_migrations'

def get_db_connection_string():
    """Return a connection string for psql."""
    return f"postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"

def ensure_version_table():
    """Create the version tracking table if it doesn't exist."""
    conn_str = get_db_connection_string()
    cmd = [
        'psql', conn_str, '-t', '-c',
        f"""
        CREATE TABLE IF NOT EXISTS {VERSION_TABLE} (
            version VARCHAR(255) PRIMARY KEY,
            applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );
        """
    ]
    subprocess.run(cmd, check=True, capture_output=True)

def get_applied_versions():
    """Return a set of already applied migration versions."""
    ensure_version_table()
    conn_str = get_db_connection_string()
    cmd = ['psql', conn_str, '-t', '-c', f"SELECT version FROM {VERSION_TABLE};"]
```

```
    result = subprocess.run(cmd, check=True, capture_output=True, text=True)
    # Parse output: one version per line, trim whitespace
    versions = {line.strip() for line in result.stdout.split('\n') if line.strip()}
    return versions

def mark_version_applied(version, direction='up'):
    """Record a migration as applied or rollback."""
    conn_str = get_db_connection_string()
    if direction == 'up':
        cmd = [
            'psql', conn_str, '-t', '-c',
            f"INSERT INTO {VERSION_TABLE} (version) VALUES ('{version}');"
        ]
    else:  # down
        cmd = [
            'psql', conn_str, '-t', '-c',
            f"DELETE FROM {VERSION_TABLE} WHERE version = '{version}';"
        ]
    subprocess.run(cmd, check=True, capture_output=True)

def run_sql_file(file_path, version):
    """Execute a SQL migration file and record its version."""
    conn_str = get_db_connection_string()
    with open(file_path, 'r') as f:
        sql_content = f.read()

    # Wrap in a transaction for atomicity (PostgreSQL)
    full_sql = f"BEGIN;\n{sql_content}\nCOMMIT;"
    cmd = ['psql', conn_str, '-t', '-c', full_sql]
    try:
        subprocess.run(cmd, check=True, capture_output=True, text=True)
        mark_version_applied(version, 'up')
        print(f"✅ Applied migration: {version}")
        return True
    except subprocess.CalledProcessError as e:
        print(f"❌ Failed to apply migration {version}: {e.stderr}")
        # Attempt to rollback the transaction
        subprocess.run(['psql', conn_str, '-t', '-c', 'ROLLBACK;'], capture_output=True)
        return False
```

D. Core Logic Skeleton Code

`database/migrate.sh` *– Main migrator orchestration script (Bash):*

```bash
#!/usr/bin/env bash

# Database Migrator: Applies pending migrations in an idempotent manner.

# Usage: ./migrate.sh [TARGET_VERSION] [DIRECTION]

#   TARGET_VERSION: Optional. If not provided, applies all unapplied migrations.

#   DIRECTION: 'up' (default) or 'down'.


set -euo pipefail


SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

MIGRATIONS_DIR="${SCRIPT_DIR}/migrations"

PY_UTILS="${SCRIPT_DIR}/utils.py"


# Import python helpers

source <(python3 "${PY_UTILS}" --export-functions 2>/dev/null || echo "echo 'Could not load utils'")


function usage() {

    echo "Usage: $0 [TARGET_VERSION] [DIRECTION]"

    echo "  TARGET_VERSION: e.g., '002_add_user_profile'"

    echo "  DIRECTION: 'up' or 'down' (default: 'up')"

    exit 1

}


function get_migration_files() {

    local direction="${1:-up}"

    find "${MIGRATIONS_DIR}" -name "*_*.${direction}.sql" | sort

}


function extract_version_from_filename() {

    local filename=$(basename "$1")

    echo "${filename}" | cut -d'_' -f1

}


function apply_migrations() {

    local target_version="${1:-}"

    local direction="${2:-up}"



    echo "Database Migrator: Applying migrations (direction: ${direction})..."



    # TODO 1: Call ensure_version_table (from utils.py) to create tracking table if needed.

    # Hint: This might involve calling a Python function or running a SQL script.



    # TODO 2: Get the list of already applied versions by calling get_applied_versions.

    # Store the result in a Bash array or variable.
```

```
    # TODO 3: Get the list of migration files for the given direction (use get_migration_files).


    # TODO 4: For each migration file in sorted order:
    #   a. Extract its version (e.g., '001' from '001_initial.up.sql').
    #   b. If a target_version is specified and this version is beyond it (lexicographically), stop.
    #   c. If direction is 'up' and the version is NOT in the applied_versions set,
    #      run the SQL file via run_sql_file and record it.
    #   d. If direction is 'down' and the version IS in the applied_versions set,
    #      run the .down.sql file and remove it from the tracking table.
    #   e. Handle errors: if any migration fails, exit the script with an error.


    # TODO 5: Print a summary of applied/rolled back migrations.


    echo "Migration process completed."
}

function validate_migration_compatibility() {
    local version_blue="$1"
    local version_green="$2"
    echo "Validating schema compatibility for app versions: Blue(${version_blue}), Green(${version_green})..."


    # TODO 1: Fetch the current schema version from the database (use get_applied_versions or similar).


    # TODO 2: Define a simple compatibility matrix or set of rules.
    # Example: "If migration 003_drop_old_column has been applied, then only app version >= 2.0 is compatible."
    # For now, we can assume all migrations are additive (expand phase) and thus always compatible.
    # In a real scenario, you might query information_schema to check for the existence of required columns.


    # TODO 3: If an incompatibility is detected (e.g., a column required by the old app is missing),
    # print an error and return a non-zero exit code.


    echo "✅ Schema is compatible with both application versions."
    return 0
}

# Main script logic
if [[ $# -gt 2 ]]; then
    usage
fi

TARGET_VERSION="${1:-}"
DIRECTION="${2:-up}"
```

```
case "${DIRECTION}" in
    up|down)
        apply_migrations "${TARGET_VERSION}" "${DIRECTION}"
        ;;
    *)
        usage
        ;;
esac
```

Example Migration Script (`database/migrations/002_add_user_profile.up.sql` – Expand Phase):

```sql
-- Expand Phase: Add new nullable column for user biography.
-- This is backward compatible; the old app ignores this column.
ALTER TABLE users ADD COLUMN bio TEXT;

-- Optional: Backfill data from an existing related table or set default values.
-- UPDATE users SET bio = '' WHERE bio IS NULL;
```

Example Migration Script (`database/migrations/003_remove_old_username.down.sql` – Contract Phase - this would be applied in a later cycle):

```sql
-- Contract Phase: Remove the old 'username' column.
-- WARNING: Only run this after confirming the new 'user_name' column is in use
-- and the old application version is no longer deployed.
ALTER TABLE users DROP COLUMN username;
```

E. Language-Specific Hints

- **Bash:** Use `set -euo pipefail` at the top of scripts to make them robust against errors. Use `[[ ... ]]` for conditional tests. When calling SQL commands, always check the exit code.
- **Python (for helpers):** The `subprocess` module is your friend for running CLI database tools. Use `check=True` to raise exceptions on failure. For more complex validation, consider using a database adapter like `psycopg2` (PostgreSQL) or `mysql-connector-python`.

F. Milestone Checkpoint After implementing the Database Migrator, you should be able to run the following validation sequence:

1. **Initial Setup:** Ensure your database is running and empty.
2. **Run First Migration:**

```bash
    cd project-root/database
    ./migrate.sh
```

`*Expected Output:* Messages indicating the `schema_migrations` table was created and migration `001_initial_schema` was applied. Ve`

3. **Simulate Expand Phase:** Add a new migration file `002_add_user_profile.up.sql` (adds a nullable `bio` column). Run the migrator again. *Expected Output:* Migration `002` is applied. The `schema_migrations` table contains two rows. The old application (if running) should not be affected.
4. **Validate Compatibility:**

```bash
    # Assume blue is v1.0 (uses old schema) and green is v1.1 (uses new bio column)
    ./migrate.sh --validate 1.0 1.1
    # Or call the function from within your deployment script
```

`*Expected Output:* "✅ Schema is compatible with both application versions."`

5. **Test Idempotency:** Run `./migrate.sh` again. It should output that no new migrations were applied (or silently succeed).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`ERROR: column "bio" already exists`	Migration script is being applied a second time (idempotency failure).	Check the `schema_migrations` table. Is the version for this migration recorded?	Ensure your `apply_migrations` function checks the version table before running a script.
Old app crashes after expand migration with `null value in column "bio" violates not-null constraint`	Added a `NOT NULL` column without a default value on existing rows.	Check the exact SQL of your migration.	Modify the migration: Add column as `NULL`, backfill data, then apply `NOT NULL` constraint, or add a `DEFAULT` value.
Rollback fails because `DROP COLUMN` migration cannot be reversed	The `.down.sql` script for a contract migration tries to recreate a dropped column with data that is lost.	Review the down script. Contract migrations are often irreversible by design.	For true rollback capability, the contract phase must be a separate, future deployment. Consider if a down script is needed (perhaps it's a no-op or logs a warning).

> **Milestone(s):** Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

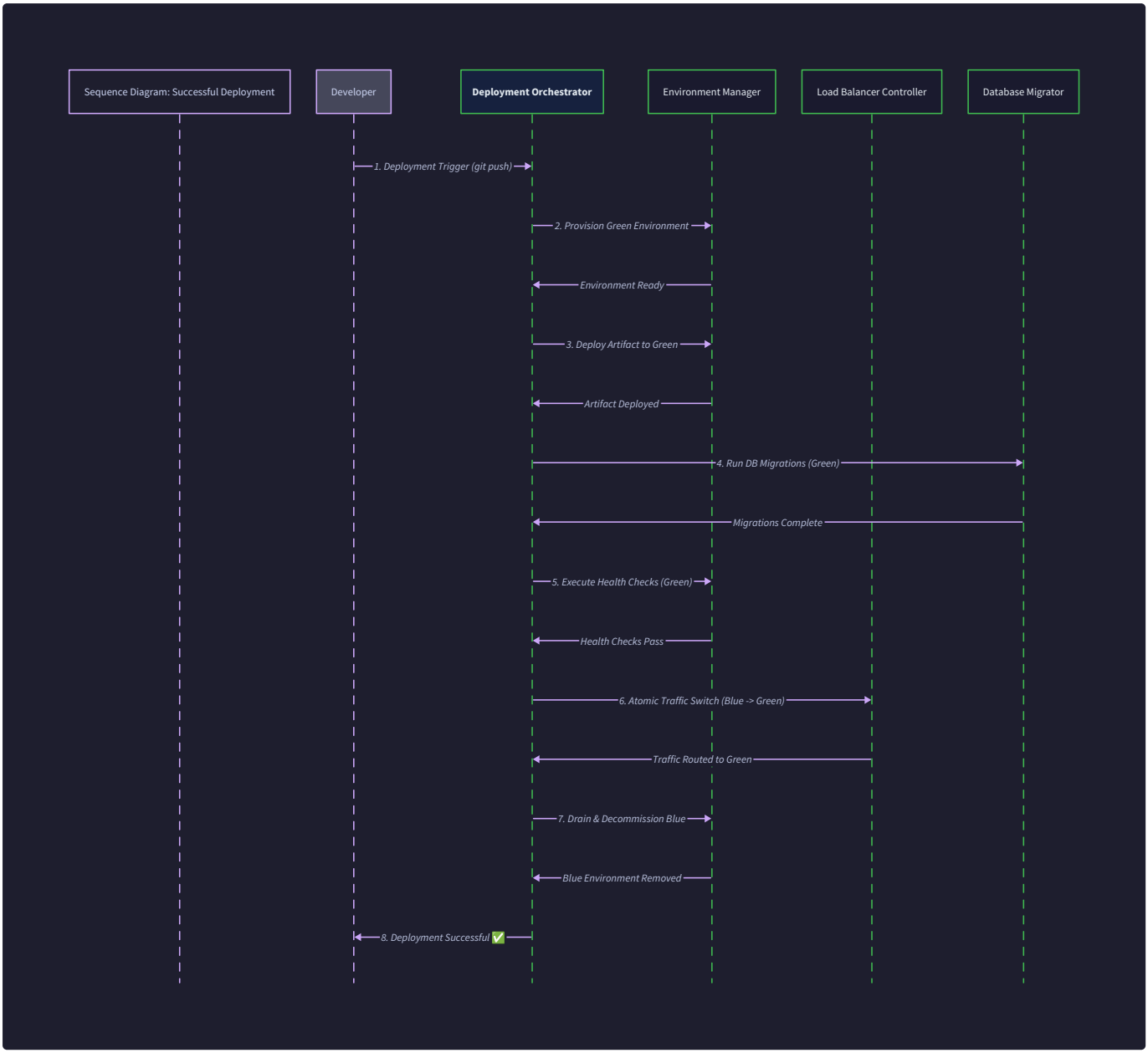
Interactions and Data Flow

This section traces the journey of a single deployment from initiation to completion and details the recovery path when things go wrong. Understanding these flows is critical to grasping how the isolated components—Environment Manager, Load Balancer Controller, Deployment Orchestrator, and Database Migrator—coordinate to achieve a seamless, zero-downtime release. The flows are designed as a series of idempotent, verifiable steps, where each step's success is a prerequisite for the next, creating a safe, automated pipeline.

Think of the deployment process as a **carefully choreographed dance**. Each dancer (component) knows their steps and timing. The choreographer (orchestrator) cues each move, but the dancers are also responsible for checking their own footing (health checks) before proceeding. If a dancer stumbles, the entire troupe can smoothly revert to the previous formation within a single beat. This dance happens while the audience (users) continues watching the performance uninterrupted, completely unaware of the complex movements behind the curtain.

Happy Path: Deploy New Version

The successful deployment of a new application version follows a strict, sequential protocol designed to minimize risk. The core principle is: **the active environment continues serving live user traffic undisturbed while the new version is prepared, validated, and finally switched into service atomically.** This flow leverages the dual-environment setup to create a staging area for the new release that is functionally identical to production.



The sequence diagram above visualizes the interaction between the system components and external actors. Below is a detailed, step-by-step walkthrough of the algorithm executed by the `Deployment Orchestrator`.

Deployment Orchestrator Algorithm: Successful Deployment

1. **Deployment Trigger**: The process begins when a developer or CI system invokes the main deployment script (e.g., `./deploy.sh v1.2.0`). The script receives the new application version tag as its primary input.
2. **State Determination**: The orchestrator calls `get_current_active_env()` to read the `ACTIVE_ENV_FILE`. This returns either `"blue"` or `"green"`. It then calls `get_inactive_environment()`, which simply returns the opposite color. This establishes the fundamental rule: *deploy to the inactive environment*.

   > **Design Insight**: Determining the inactive environment by reading persistent state, rather than inferring from runtime process lists, ensures the deployment remains idempotent and safe to rerun after any partial failure.

3. **Pre-Flight Health Check**: Before any changes, the orchestrator calls `check_health(active_env)` on the currently active environment. This is a deep health check, verifying the application's HTTP endpoints, database connectivity, and any downstream service dependencies. If this check fails, the deployment aborts immediately, as the system's starting state is considered unstable.
4. **Database Migration Compatibility Check**: The orchestrator consults the `Database Migrator` by calling `validate_migration_compatibility(app_version_active, new_version)`. This function compares the schema requirements of the currently running application version with the new version. It verifies that the database's current schema (via `get_current_schema_version()`) is compatible with **both** versions. If the new version requires a forward migration, it must be done using the **expand-contract pattern** to maintain backward compatibility. If validation fails (e.g., a breaking change without an expand phase), deployment halts.

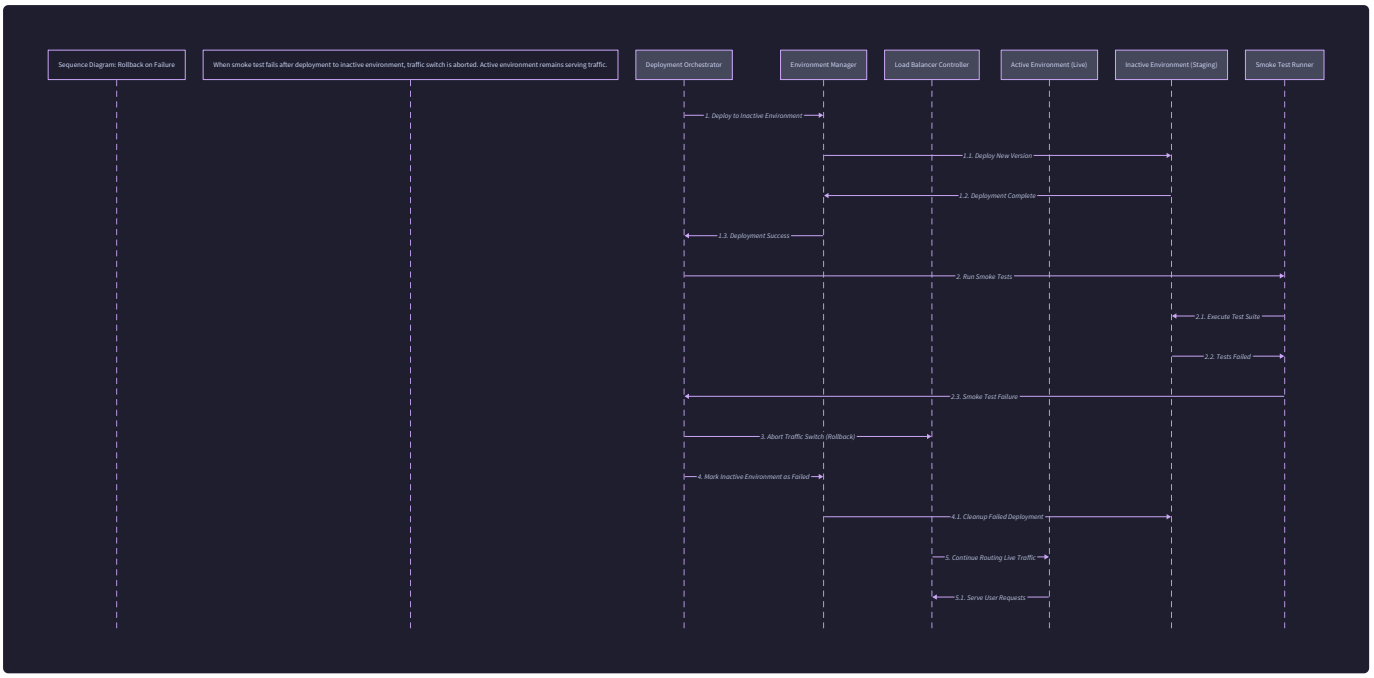
5. **Environment Preparation**: The orchestrator calls `start_environment(inactive_env)` on the `Environment Manager`. This bootstraps the inactive environment, which may have been stopped after a previous deployment. The manager injects the correct `ENVIRONMENT_COLOR` and port (e.g., `GREEN_PORT=8081`) via environment variables, pulls the new version's Docker image, and starts the containerized application.
6. **Deployment to Inactive Environment**: With the environment running, the orchestrator executes `deploy_to_environment(inactive_env, new_version)`. This function is responsible for any environment-specific deployment tasks, such as copying configuration files, setting up secrets, or notifying sidecar services. The application in the inactive environment now runs the new code, isolated from user traffic.
7. **Post-Deployment Health Validation**: The orchestrator now calls `check_health(inactive_env)` on the newly deployed environment. This check is more rigorous and may include:
    - Verifying the application process is running.
    - Ensuring the health endpoint (`/health`) returns a `200 OK` with `"status": "healthy"`.
    - Confirming the `version` field in the health response matches the new version tag.
    - Validating database connectivity (`database_connected: true`). The deployment cannot proceed until the inactive environment passes this health gate.
8. **Smoke Test Execution**: After health checks pass, the orchestrator runs `run_smoke_tests(inactive_env, port)`. This suite of automated tests validates the critical user journeys and API endpoints of the new version. Tests are run *against the inactive environment's specific port/URL*, ensuring no interference with live traffic. Smoke tests might verify login, search, checkout, or other core business workflows. A `true` return value indicates all tests passed.
9. **Traffic Switch Preparation**: The orchestrator now instructs the `Load Balancer Controller` to prepare for the cutover. It calls `generate_config_for_env(inactive_env)` to produce the new load balancer configuration that points the primary upstream to the now-healthy inactive environment. It then calls `validate_load_balancer_config()` to perform a syntax check on the new configuration file, ensuring it won't break the load balancer.
10. **Connection Draining & Traffic Switch**: This is the **atomic moment** of the deployment. The orchestrator calls `switch_traffic(inactive_env)`. Internally, this function:
    - Updates the `ACTIVE_ENV_FILE` to the new environment color.
    - Triggers a **graceful reload** of the load balancer (e.g., `nginx -s reload` or `haproxy -sf`). This command loads the new configuration, starts new worker processes with the new routing rules, and gracefully terminates old workers after they finish serving existing (in-flight) connections. This achieves **zero-downtime** and **connection draining**.
    - Verifies that the load balancer's own health check starts passing for the new active environment.
11. **Post-Switch Verification**: After the switch, a final health check `check_health(new_active_env)` is performed *through the load balancer's public endpoint*. This confirms the new environment is correctly receiving and serving traffic. The orchestrator may also run a subset of smoke tests against the public URL to ensure full integration.
12. **Cleanup (Optional)**: The old active environment (now inactive) may be stopped by calling `stop_environment(old_active_env)` to free up resources. However, it is often kept running for a period to enable instantaneous rollback.
13. **Logging and Notification**: Every step's outcome is logged to `DEPLOYMENT_LOG`, and a final notification is sent (e.g., to Slack, email) indicating the successful deployment of `new_version` to `new_active_env`.

Concrete Walk-Through Example Consider a system where `blue` is currently active (serving on port 8080) and `green` is inactive (stopped). The new version is `v2.5.1`.

1. Trigger: `./deploy.sh v2.5.1`
2. State: `get_current_active_env()` returns `"blue"`. `get_inactive_environment()` returns `"green"`.
3. Pre-flight: Health check on `blue` passes.
4. DB Check: `validate_migration_compatibility("v2.4.0", "v2.5.1")` passes; v2.5.1 only uses existing columns.
5. Prep: `start_environment("green")` starts the green app on port 8081 with `v2.5.1`.
6. Deploy: `deploy_to_environment("green", "v2.5.1")` configures the app.
7. Health: `check_health("green")` passes. The `/health` endpoint reports `{"environment": "green", "version": "v2.5.1", ...}`.
8. Smoke: `run_smoke_tests("green", 8081)` passes.
9. Prep Config: New Nginx config is generated with `upstream active_backend { server localhost:8081; }` and validated.
10. Switch: `switch_traffic("green")` updates the state file to `"green"` and executes `nginx -s reload`. User traffic now flows to port 8081.
11. Verify: A `curl http://localhost/health` returns data from the green environment, confirming the switch.
12. Cleanup: `stop_environment("blue")` is called.
13. Log: Entry written: `"SUCCESS: Deployed v2.5.1 to green, switched traffic from blue at $(date)"`.

Rollback Path

Despite thorough testing, defects can surface only under production traffic. The blue-green architecture's most powerful feature is its **instant rollback** capability. If the new version exhibits critical failures after deployment to the inactive environment (or even after the traffic switch), the system can revert to the known-good previous version within seconds by simply switching traffic back.



The rollback path is a disciplined emergency procedure. Think of it as a **fire alarm system in a building**. When a smoke detector (smoke test) goes off, the pre-defined evacuation plan (rollback script) is triggered automatically. People (user traffic) are directed to the safe, known exit (old environment) without panic or delay, while the fire department (developers) can investigate the affected area (failed new version) without putting anyone at risk.

Rollback Triggers Rollbacks can be initiated automatically or manually:

- **Automatic**: Failure of the `run_smoke_tests()` function after deployment to the inactive environment.
- **Automatic**: Failure of the post-switch verification health check.
- **Manual**: Operations team observes elevated error rates or critical issues via monitoring and invokes the rollback command.

Deployment Orchestrator Algorithm: Rollback on Smoke Test Failure

This flow details the most common automatic rollback scenario, where a problem is detected *before* the traffic switch.

1. **Failure Detection**: Steps 1-7 of the Happy Path complete successfully. However, during Step 8, `run_smoke_tests(inactive_env, port)` returns `false`. The test suite has identified a critical failure in the new version.
2. **Rollback Initiation**: The orchestrator immediately aborts the deployment sequence and calls `rollback_deployment(failed_env)`, passing the color of the environment that just failed the tests (the inactive environment).
3. **State Assessment**: The `rollback_deployment` function first reads `get_current_active_env()` to confirm which environment is still serving live traffic. This should still be the original active environment, as the traffic switch never occurred.
4. **Validation of Rollback Target**: A health check `check_health(original_active_env)` is performed on the still-active environment. This is crucial to ensure the rollback target is itself healthy. If it is unhealthy, a rollback may be impossible, and an alert is raised for immediate human intervention.
5. **Traffic Switch Guarantee**: Since the traffic was never switched, the load balancer is already pointing to the healthy, original environment. Therefore, **no actual traffic switch is needed**. The rollback function's primary action is to leave the `ACTIVE_ENV_FILE` unchanged and to log the aborted deployment.
6. **Failure Cleanup**: The orchestrator then calls `stop_environment(failed_env)` on the environment that failed the smoke tests. This shuts down the faulty version to free up resources and prevent any accidental access.
7. **Alerting and Logging**: A high-priority alert is sent to the development and operations teams. A detailed entry is made in `DEPLOYMENT_LOG`, including the version that failed, the reason (smoke test failure), and the fact that traffic was not switched.

> **Design Insight**: The rollback before a switch is a "no-op" for traffic but critical for system hygiene. It prevents a bad version from lingering and ensures the pipeline state is clean for the next deployment attempt.

Rollback *After* a Traffic Switch A more severe scenario is when a defect is discovered *after* the traffic switch has occurred (e.g., via post-switch monitoring). The rollback procedure is slightly different but equally fast.

1. **Trigger**: Monitoring alerts on error rates, or manual observation triggers a rollback.
2. **State Read**: `get_current_active_env()` returns the newly activated environment (e.g., `green`), which is now problematic.
3. **Determine Previous Environment**: The orchestrator infers the previous environment (e.g., `blue`). This can be derived from a deployment history log or by reading a backup of the previous `ACTIVE_ENV_FILE`.

4. **Health Check**: `check_health(previous_env)` is called. If the old environment is still running (which is recommended practice), it should pass.
5. **Traffic Switch Back**: The orchestrator calls `switch_traffic(previous_env)`. This performs the same atomic, graceful reload operation, but in the reverse direction, routing traffic back to the previous environment.
6. **Log and Alert**: The rollback is logged, and teams are notified of the production rollback.

Common Pitfalls in Rollback Flows

⚠️ Pitfall: Assuming the Previous Environment is Healthy

- **Description**: Automatically switching traffic back without verifying the health of the old environment.
- **Why it's Wrong**: If the old environment has crashed or developed issues since being taken out of service (e.g., due to a latent memory leak), rolling back to it will cause an immediate production outage, moving from one broken state to another.
- **How to Fix**: Always call `check_health()` on the rollback target environment before executing `switch_traffic()`.

⚠️ Pitfall: Not Preserving the Old Environment

- **Description**: Immediately stopping and destroying the old active environment after a successful traffic switch.
- **Why it's Wrong**: Eliminates the possibility of an instant rollback. If a bug is found minutes after the switch, you have nowhere safe to revert.
- **How to Fix**: Implement a waiting period (e.g., 1 hour) or a manual approval step before cleaning up the old environment. Better yet, keep it stopped but available to be restarted quickly if needed.

⚠️ Pitfall: Complex, Multi-Step Rollbacks

- **Description**: Designing a rollback that requires undoing database migrations, reversing configuration changes, and redeploying old artifacts.
- **Why it's Wrong**: Slow and error-prone. The "instant" promise of blue-green is broken.
- **How to Fix**: Adhere strictly to backward-compatible **expand-contract migrations** and immutable artifacts. Rollback should *only* be a traffic switch. The old environment's code and data schema must remain fully operational.

Implementation Guidance

This section provides the concrete scripts and scaffolding to implement the data flows described above. The primary language is Bash, chosen for its simplicity and universal availability in CI/CD environments.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Orchestration Script	Monolithic Bash script with functions	Makefile or Taskfile with discrete, composable tasks
State Management	Flat file (`ACTIVE_ENV_FILE`)	Key-value store (Redis, Consul) or configuration service
Smoke Testing	`curl` commands in Bash	Dedicated testing framework (Pytest, Jest) called from script
Logging	Append to text file (`DEPLOYMENT_LOG`)	Structured logging (JSON) to stdout, collected by a log aggregator (Loki, ELK)

B. Recommended File/Module Structure Extend the existing project structure to house the orchestration logic and state files:

```
blue-green-project/
├── scripts/
│   ├── deploy.sh                  # Main deployment orchestrator (this section)
│   ├── rollback.sh                # Dedicated rollback script
│   ├── lib/
│   │   ├── environment_manager.sh   # Functions: start_environment, check_health, etc.
│   │   ├── load_balancer_controller.sh # Functions: switch_traffic, generate_config, etc.
│   │   └── database_migrator.sh     # Functions: apply_migrations, validate_compatibility, etc.
│   └── smoke_tests/
│       └── run.sh                 # Entry point for smoke test suite
├── config/
│   └── loadbalancer/
│       └── nginx.conf.template      # Template for load balancer config
├── state/                         # Directory for state files (gitignored)
│   ├── active_environment         # The ACTIVE_ENV_FILE
│   └── deployment.log             # The DEPLOYMENT_LOG
└── app/                           # Your application code
    └── ...
```

C. Infrastructure Starter Code Here is a complete, reusable bash library for logging and state management that your orchestrator can source.

`scripts/lib/utils.sh`:

BASH

```bash
#!/bin/bash

# Common utilities for deployment scripts

# Configuration
readonly ACTIVE_ENV_FILE="/tmp/active_environment"
readonly DEPLOYMENT_LOG="/var/log/deployments.log"
readonly BLUE_PORT=8080
readonly GREEN_PORT=8081

# Log a message with timestamp to file and stdout
log() {
    local level="$1"
    local message="$2"
    local timestamp
    timestamp=$(date '+%Y-%m-%d %H:%M:%S')
    echo "[${timestamp}] ${level}: ${message}" | tee -a "$DEPLOYMENT_LOG"
}

# Read the currently active environment from file
get_current_active_env() {
    if [[ -f "$ACTIVE_ENV_FILE" ]]; then
        cat "$ACTIVE_ENV_FILE"
    else
        # Initialize if file doesn't exist
        echo "blue" > "$ACTIVE_ENV_FILE"
        echo "blue"
    fi
}

# Write the active environment to file
set_current_active_env() {
    local env="$1"
    echo "$env" > "$ACTIVE_ENV_FILE"
    log "INFO" "Updated active environment to: $env"
}

# Determine the inactive environment
get_inactive_environment() {
    local active_env
    active_env=$(get_current_active_env)
    if [[ "$active_env" == "blue" ]]; then
        echo "green"
    else
```

```
        echo "blue"
    fi
}

# Get the port for a given environment
get_port_for_env() {
    local env="$1"
    if [[ "$env" == "blue" ]]; then
        echo "$BLUE_PORT"
    else
        echo "$GREEN_PORT"
    fi
}

# Check if a command exists, log error and exit if not
require_command() {
    local cmd="$1"
    if ! command -v "$cmd" &> /dev/null; then
        log "ERROR" "Required command '$cmd' is not installed. Aborting."
        exit 1
    fi
}
```

D. Core Logic Skeleton Code The main deployment script implements the step-by-step algorithm described in the Happy Path.

`scripts/deploy.sh`:

```bash
#!/bin/bash

# Main Deployment Orchestrator Script

# Usage: ./deploy.sh <version_tag>


set -euo pipefail  # Exit on error, undefined variable, and pipe failure


# Source utility libraries

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

source "${SCRIPT_DIR}/lib/utils.sh"

source "${SCRIPT_DIR}/lib/environment_manager.sh"

source "${SCRIPT_DIR}/lib/load_balancer_controller.sh"

source "${SCRIPT_DIR}/lib/database_migrator.sh"


# --- Configuration ---

VERSION_TAG="$1"

SERVICE_NAME="myapp"


# --- Main Deployment Function ---

main() {

    log "INFO" "Starting deployment of version: $VERSION_TAG"


    # Step 1 & 2: Determine environments

    local active_env inactive_env

    active_env=$(get_current_active_env)

    inactive_env=$(get_inactive_environment)

    log "INFO" "Active: $active_env, Inactive: $inactive_env"


    # Step 3: Pre-flight health check on active environment

    log "INFO" "Performing pre-flight health check on active environment ($active_env)..."

    # TODO 1: Call check_health for the active_env. If it fails, log error and exit.

    # Hint: Use the function from environment_manager.sh


    # Step 4: Database migration compatibility check

    log "INFO" "Validating database migration compatibility..."

    # TODO 2: Call validate_migration_compatibility with current app version and new version.

    # If validation fails, log error and exit.


    # Step 5 & 6: Prepare and deploy to inactive environment

    log "INFO" "Starting and deploying to inactive environment ($inactive_env)..."

    # TODO 3: Call start_environment for the inactive_env.

    # TODO 4: Call deploy_to_environment for the inactive_env with the VERSION_TAG.


    # Step 7: Post-deployment health check on inactive environment

    log "INFO" "Waiting for health check on $inactive_env..."

    # TODO 5: Implement a retry loop (e.g., 5 attempts with 5-second sleep) calling check_health on inactive_env.
```

```
    # If health check fails after retries, trigger rollback (call rollback_deployment).

    # Step 8: Run smoke tests
    local inactive_port
    inactive_port=$(get_port_for_env "$inactive_env")
    log "INFO" "Running smoke tests against $inactive_env (port $inactive_port)..."
    # TODO 6: Call run_smoke_tests for the inactive_env and port.
    # If smoke tests fail, trigger rollback (call rollback_deployment).

    # Step 9 & 10: Prepare config and switch traffic
    log "INFO" "Switching traffic to $inactive_env..."
    # TODO 7: Call switch_traffic for the inactive_env.
    # This function should handle config generation, validation, and graceful reload.

    # Step 11: Post-switch verification
    log "INFO" "Verifying deployment through load balancer..."
    # TODO 8: Perform a final health check via the public load balancer endpoint (e.g., curl http://localhost/health).
    # Verify the response contains the new version.

    log "SUCCESS" "Deployment of $VERSION_TAG to $inactive_env completed successfully."
}

# --- Rollback Function (called on failure) ---
rollback_deployment() {
    local failed_env="$1"
    log "ERROR" "Deployment failed. Initiating rollback for environment: $failed_env"
    local active_env
    active_env=$(get_current_active_env)

    # If the traffic hasn't been switched yet, the active_env is still the good one.
    if [[ "$failed_env" == "$active_env" ]]; then
        log "WARN" "Failed environment is currently active. Manual intervention required."
        exit 1
    fi

    # Ensure the other environment (the one we might roll back to) is healthy
    local rollback_target="$active_env"  # This is still the original active env
    log "INFO" "Checking health of rollback target ($rollback_target)..."
    # TODO 9: Call check_health on rollback_target. If it fails, log error and exit (manual intervention needed).

    # Since traffic is still pointing to rollback_target, no switch is needed.
    log "INFO" "Traffic is already on $rollback_target. Stopping failed environment $failed_env..."
    # TODO 10: Call stop_environment on the failed_env.
    log "INFO" "Rollback complete. System is stable on $rollback_target."
```

```
    exit 1
}

# Trap errors and call rollback (for failures after deployment to inactive env but before switch)
trap 'rollback_deployment $inactive_env' ERR

main "$@"
```

E. Language-Specific Hints

- **Error Handling**: Use `set -euo pipefail` at the top of your Bash scripts to make them robust. Combine with `trap` for cleanup on error.
- **Idempotency**: Design all functions (like `start_environment`) to be safe to run multiple times. Check if an environment is already running before attempting to start it.
- **Retry Logic**: Implement simple retry loops for health checks using `while` loops and `sleep`.
- **Input Validation**: Always validate the version tag argument and ensure it follows your expected format.
- **Logging**: Use `tee -a` to log to both stdout and a file simultaneously for audit trails.

F. Milestone Checkpoint After implementing the `deploy.sh` script and its supporting libraries, you should be able to run a dry-run or full deployment.

1. **Setup**: Ensure your blue and green environments are set up (Milestone 1) and the load balancer is routing to blue (Milestone 2).
2. **Test Command**: Run `./scripts/deploy.sh v1.0.0-test` (use a dummy version tag).
3. **Expected Behavior**:
   - The script identifies blue as active, green as inactive.
   - It starts/stops the green environment.
   - It runs health checks and (if you have simple smoke tests) runs them against green.
   - Finally, it switches traffic to green and verifies.
   - Check the `DEPLOYMENT_LOG` for a success message.
4. **Verification**:
   - Run `curl http://localhost/health` repeatedly during the process. You should never receive a connection refused or 5xx error.
   - After the switch, the health endpoint should show `"environment": "green"` and the new version.
5. **Signs of Trouble**:
   - **Health check fails**: Verify the environment's app is running and the health endpoint is accessible on the correct port.
   - **Traffic not switching**: Check the `ACTIVE_ENV_FILE` and the load balancer configuration file. Ensure the `switch_traffic` function is calling `nginx -s reload` and not `nginx -s restart`.
   - **Script exits midway**: Check the error log. Use `bash -x ./scripts/deploy.sh v1.0.0` to run in debug mode and see each command.

Error Handling and Edge Cases

> **Milestone(s):** Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

Even the most carefully designed system will encounter failures. The true test of a blue-green deployment architecture isn't whether failures occur—they inevitably will—but how gracefully the system detects, responds to, and recovers from them. This section catalogs the failure modes you must anticipate and the recovery procedures that transform potential disasters into managed incidents.

Think of this system as a **fire department for your deployment pipeline**. Just as a fire department maintains equipment, trains firefighters, and establishes protocols for different types of emergencies, this system needs predefined detection mechanisms and recovery procedures for each failure category. When smoke appears (a failed health check), the right team springs into action with the right tools, following practiced procedures to contain the damage.

Failure Categories and Recovery

Failures in a blue-green deployment system fall into distinct categories, each requiring specific detection strategies and recovery procedures. The table below organizes these failures by the component most responsible for detection and the appropriate recovery action.

Failure Mode	Primary Detector	Detection Strategy	Recovery Procedure	Time to Recovery
Deployment Failure (artifact build fails, container won't start)	Environment Manager	Exit code monitoring from `deploy_to_environment()`; container health check timeout	Abort deployment sequence; keep existing environment active; alert engineers	Immediate (seconds)
Health Check Failure (environment becomes unhealthy after deployment)	Load Balancer Controller	Periodic `check_health()` returns false; load balancer health checks fail	Prevent traffic switch to unhealthy environment; if already switched, trigger `rollback_deployment()`	< 30 seconds
Smoke Test Failure (critical functionality broken in new version)	Deployment Orchestrator	`run_smoke_tests()` returns false for deployed environment	Trigger `rollback_deployment()` before traffic switch; alert with detailed test results	< 60 seconds
Traffic Switch Failure (load balancer config error, reload fails)	Load Balancer Controller	`switch_traffic()` returns error; `validate_load_balancer_config()` fails	Roll back to previous config; keep traffic on current environment; alert with config validation errors	< 10 seconds
Database Migration Failure (schema change fails, breaks compatibility)	Database Migrator	`apply_migrations()` returns false; `validate_migration_compatibility()` fails	Roll back migration if possible; block deployment; keep both environments on previous version	Varies (minutes to hours)
Partial Deployment (new version deployed but old containers not cleaned up)	Environment Manager	Resource monitoring detects excess containers/processes	Clean up orphaned resources; verify only one instance per environment color	During next deployment cycle
Configuration Drift (blue and green environments develop differences)	Environment Manager	Configuration hash comparison; environment inventory audit	Re-sync configurations; redeploy affected environment; update Infrastructure as Code templates	Next maintenance window
Resource Exhaustion (memory, CPU, disk space limits reached)	Infrastructure Monitoring	System metrics thresholds exceeded; container evictions	Scale infrastructure; clean up logs/temporary files; optimize resource requests	Minutes to hours
Network Partition (environment cannot reach database or dependencies)	Environment Manager	`check_health()` detects `database_connected` = false	Mark environment unhealthy; avoid traffic switch; investigate network connectivity	Until partition resolved
State Corruption (`ACTIVE_ENV_FILE` contains invalid value or is missing)	Load Balancer Controller	`get_current_active_env()` returns invalid color or cannot read file	Use conservative default (keep current traffic routing); manual intervention required	Manual recovery needed

Detailed Recovery Procedures

Deployment Failure Recovery: When `deploy_to_environment()` fails (non-zero exit code), the Deployment Orchestrator immediately aborts the deployment sequence. It logs the failure to `DEPLOYMENT_LOG` with the exact error message and exit code. No traffic switch occurs, and the system remains in its previous stable state. This is the **safest possible failure**—the new version never reaches users.

Health Check Failure Recovery: The Load Balancer Controller performs health checks every 5 seconds. When three consecutive checks fail, the environment is marked "unhealthy." If this occurs:

- **Before traffic switch:** The `switch_traffic()` function refuses to proceed and returns an error.
- **After traffic switch:** The controller immediately calls `rollback_deployment()` to revert traffic to the previous environment, then alerts engineers with health check details.

Smoke Test Failure Recovery: The Deployment Orchestrator runs `run_smoke_tests()` against the newly deployed (inactive) environment. If any critical test fails:

1. Log test failure details to `DEPLOYMENT_LOG`
2. Immediately call `rollback_deployment()` to revert any partial changes
3. Send alert with specific failing endpoint and expected vs. actual response
4. Leave the failed environment running for debugging (with appropriate labeling)

Traffic Switch Failure Recovery: When `switch_traffic()` encounters an error (e.g., Nginx config syntax error, permission denied), it:

1. Attempts to reload the previous known-good configuration
2. Verifies the load balancer is still serving traffic (even if to the "wrong" environment)
3. Logs the exact error and preserves the failed configuration for debugging
4. Alerts engineers with the validation error output

> **Critical Insight:** The worst traffic switch failure is a **silent failure** where the configuration appears to apply but doesn't take effect. Always verify the switch by checking `get_current_active_env()` matches the expected environment after the switch command returns successfully.

Database Migration Failure Recovery: Database failures require the most careful handling. The Database Migrator employs a multi-stage recovery:

1. **Forward migration fails:** The migrator attempts to roll back any partially applied changes using migration rollback scripts. If rollback succeeds, deployment halts. If rollback fails, manual database intervention is required.
2. **Backward compatibility check fails:** Deployment is blocked before any environment deployment occurs. Both blue and green must remain compatible with the current schema.
3. **Post-switch data inconsistency detected:** Consider triggering a rollback even if the application appears healthy, as data corruption may not be immediately apparent.

Edge Cases

Edge cases are unusual scenarios that test the boundaries of your system's resilience. Unlike routine failures, these situations often involve multiple simultaneous failures or rare environmental conditions.

Simultaneous Deployments

Scenario: Two engineers trigger deployments at nearly the same time, or an automated deployment starts while another is in progress.

Detection: The Deployment Orchestrator implements a **deployment lock** mechanism. Before starting any deployment, it acquires an exclusive lock (file lock on `/tmp/deployment.lock`). If the lock is already held:

Detection Method	Action
Lock acquisition fails	Current deployment waits (with timeout) or fails immediately with clear error
Deployment already in progress	Second deployment receives "deployment in progress" status and can either queue or abort
Stale lock (older than timeout)	Force release the lock after validation, log warning about possible orphaned deployment

Recovery: Implement idempotent deployment operations so that if two deployments somehow proceed, the second one's operations have no harmful effect (e.g., deploying the same version to an environment already running that version is harmless).

Network Partitions (Split-Brain Scenario)

Scenario: A network partition separates the blue environment from the database but not the green environment, or separates the deployment control plane from one of the environments.

Mental Model: Imagine **air traffic control losing radio contact with one runway** while planes continue to land. The controller must assume the worst and divert all traffic to the runway they can still communicate with.

Detection Strategies:

1. **Environment Manager health checks** include database connectivity (`database_connected` field). A partition causing database disconnection is detected within health check interval.
2. **Control plane partition** (orchestrator cannot reach environment) detected via deployment operation timeouts.
3. **Asymmetric partition** (load balancer can reach one environment but not the other) detected via load balancer health checks.

Recovery Procedures:

Partition Type	Detection	Recovery Action
Environment→Database	`database_connected` = false in health check	Mark environment unhealthy; prevent traffic switch to it
Orchestrator→Environment	`deploy_to_environment()` times out	Abort deployment; manual verification required
Load Balancer→Environment	Load balancer health checks fail	Automatically remove environment from upstream pool
Complete split-brain	Conflicting health status reports from different observers	Manual intervention; conservative approach: don't switch traffic

Prevention: Use redundant network paths and implement circuit breakers in the application to fail fast when dependencies are unavailable.

Partial Failures During Rollback

Scenario: A rollback starts (due to smoke test failure) but encounters its own failure during execution.

Detection: The `rollback_deployment()` function includes its own health checks and validation:

1. Verify the previous environment is still healthy before switching traffic back
2. Validate load balancer configuration for the rollback target

3. Confirm the rollback completes successfully

Recovery Procedure for Failed Rollback:

1. **If traffic cannot be switched back:** The system remains in a degraded state (traffic on failing environment). Alert engineers immediately with high priority. Provide manual recovery steps.
2. **If previous environment is also unhealthy:** This is a **double failure** scenario. The system should:
   - Attempt to restart the previous environment using `start_environment()`
   - If restart succeeds, switch traffic to it
   - If restart fails, alert for immediate manual intervention
3. **If rollback script itself fails:** Log detailed error, preserve system state for debugging, and alert for manual recovery.

> **Design Principle:** A failed rollback is more dangerous than the original deployment failure because it leaves the system in a known-bad state. Rollback procedures must be simpler and more reliable than deployment procedures.

Database Migration Edge Cases

Backward Compatibility Violation: The new application version works with the new schema but the old version breaks.

Detection: `validate_migration_compatibility()` simulates both application versions against the current schema.

Recovery: Block deployment until migrations are adjusted. If already deployed, immediate rollback is required.

Expand-Contract Timing Issues: The "contract" phase of a migration (dropping old column, adding NOT NULL constraint) is applied while old application versions are still running.

Detection: Monitoring queries fail on old application pods after contract migration.

Recovery: Revert the contract migration (re-add nullable column if dropped, remove constraint) and wait for all old pods to be replaced.

Data Corruption During Migration: Migration script has a bug that corrupts existing data.

Detection: Data validation checks in migration scripts; monitoring of data quality metrics.

Recovery: Restore from backup if corruption is extensive; use migration rollback script if possible.

State Corruption Scenarios

Corrupted `ACTIVE_ENV_FILE`: File contains invalid value ("purple"), is empty, or is missing.

Detection: `get_current_active_env()` validates the value is either "blue" or "green".

Recovery: Conservative approach: assume current traffic routing is correct (check load balancer actual configuration) and rewrite file with correct value. Alert for investigation.

Stale Environment Resources: Old containers, volumes, or network configurations from previous deployments not cleaned up.

Detection: Resource audit during deployment preparation; monitoring of resource counts.

Recovery: Cleanup script runs before each deployment; orphan resource collector runs periodically.

Human Error Scenarios

Manual Intervention Gone Wrong: Engineer manually modifies load balancer configuration or environment directly, bypassing automation.

Detection: Configuration drift detection; checksums of managed files.

Recovery: Re-apply automation to resync state; implement read-only protections for production infrastructure.

Incorrect Rollback Trigger: Automated monitoring falsely detects a failure and triggers unnecessary rollback.

Detection: Rollback reason logging; verification of failure before rollback.

Recovery: Implement "circuit breaker" pattern for automated rollbacks—too many rollbacks in a short period should trigger investigation rather than another rollback.

Implementation Guidance

While error handling design is conceptual, implementation makes it concrete. Below you'll find starter code for critical error handling components.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Detection	Exit code checking + health endpoints	Distributed tracing + anomaly detection
Alerting	Email/Slack notifications	PagerDuty/OpsGenie integration with escalation
State Recovery	File-based locks and state	Distributed consensus (etcd, ZooKeeper)
Circuit Breakers	Simple counter in bash scripts	Resilience patterns library (Hystrix, resilience4j)

Recommended File Structure

```
blue-green-system/
├── scripts/
│   ├── error_handlers.sh           # Common error handling functions
│   ├── deployment_lock.sh          # Lock management for deployments
│   └── cleanup_orphans.sh          # Clean up stale resources
├── config/
│   └── alert_rules.yaml            # Alert conditions and thresholds
└── logs/
    ├── deployments.log             # Main deployment log (DEPLOYMENT_LOG)
    └── errors/                     # Detailed error dumps per incident
```

Infrastructure Starter Code

Error Handler Library (scripts/error_handlers.sh):

```bash
#!/usr/bin/env bash

# Common error handling functions for blue-green deployment system

set -o errexit   # Exit on any command failure

set -o nounset   # Error on unset variables

set -o pipefail  # Pipeline fails if any command fails

# Log error with timestamp and context

log_error() {

    local context="$1"

    local message="$2"

    local timestamp=$(date -u +"%Y-%m-%dT%H:%M:%SZ")

    echo "[ERROR][$timestamp][$context] $message" | tee -a "${DEPLOYMENT_LOG:-/var/log/deployments.log}"

}

# Send alert notification (simple version)

send_alert() {

    local severity="$1"

    local subject="$2"

    local body="$3"

    # Simple email alert (configure SSMTP or similar)

    echo -e "Subject: [$severity] $subject\n\n$body" | sendmail -t alerts@example.com 2>/dev/null || true

    # Also log as critical error

    log_error "ALERT:$severity" "$subject: $body"

}

# Check if environment is healthy with retries

check_health_with_retry() {

    local env_color="$1"

    local max_retries="${2:-3}"

    local retry_delay="${3:-5}"

    for ((i=1; i<=max_retries; i++)); do

        if check_health "$env_color"; then

            return 0

        fi

        if [[ $i -lt $max_retries ]]; then

            log_error "health_check" "Health check failed for $env_color (attempt $i/$max_retries), retrying in ${retry_delay}s..."

            sleep "$retry_delay"

        fi
```

```
    done

    log_error "health_check" "Health check failed for $env_color after $max_retries attempts"
    return 1
}

# Acquire deployment lock with timeout
acquire_deployment_lock() {
    local lock_file="/tmp/deployment.lock"
    local timeout_seconds="${1:-300}"  # 5 minute default timeout

    # Check for stale lock (older than 1 hour)
    if [[ -f "$lock_file" ]]; then
        local lock_age=$(($(date +%s) - $(stat -c %Y "$lock_file" 2>/dev/null || echo 0)))
        if [[ $lock_age -gt 3600 ]]; then
            log_error "lock_manager" "Found stale lock file (${lock_age}s old), removing..."
            rm -f "$lock_file"
        fi
    fi

    # Try to acquire lock with timeout
    local end_time=$((SECONDS + timeout_seconds))
    while [[ $SECONDS -lt $end_time ]]; do
        if (set -o noclobber; echo "$$" > "$lock_file") 2>/dev/null; then
            trap 'release_deployment_lock' EXIT
            log_error "lock_manager" "Acquired deployment lock (PID: $$)"
            return 0
        fi
        sleep 5
    done

    log_error "lock_manager" "Failed to acquire deployment lock after ${timeout_seconds}s"
    return 1
}

# Release deployment lock
release_deployment_lock() {
    local lock_file="/tmp/deployment.lock"
    if [[ -f "$lock_file" ]] && [[ $(cat "$lock_file") == "$$" ]]; then
        rm -f "$lock_file"
        log_error "lock_manager" "Released deployment lock"
```

```
    fi
}

# Validate active environment file contents
validate_active_env_file() {
    local active_env_file="${ACTIVE_ENV_FILE:-/tmp/active_environment}"

    if [[ ! -f "$active_env_file" ]]; then
        log_error "state_validation" "Active environment file not found: $active_env_file"
        return 1
    fi

    local active_env=$(cat "$active_env_file" | tr -d '[:space:]')
    if [[ "$active_env" != "blue" ]] && [[ "$active_env" != "green" ]]; then
        log_error "state_validation" "Invalid active environment value: '$active_env'"
        return 1
    fi

    echo "$active_env"
    return 0
}

# Emergency rollback to specified environment
emergency_rollback() {
    local target_env="$1"
    local reason="$2"

    log_error "EMERGENCY_ROLLBACK" "Initiating emergency rollback to $target_env: $reason"

    # TODO 1: Verify target environment is healthy
    # TODO 2: Force traffic switch to target environment
    # TODO 3: Send high-priority alert about emergency rollback
    # TODO 4: Document rollback reason and context for post-mortem
}
```

Core Logic Skeleton Code

Deployment Orchestrator with Error Handling (scripts/deploy.sh):

```bash
#!/usr/bin/env bash

# Main deployment orchestrator with comprehensive error handling

main() {
    local version="$1"

    # Set up error handling
    set -eE  # Exit on error, enable error tracing
    trap 'deployment_failed "Unhandled error at line $LINENO"' ERR
    trap 'cleanup_temp_resources' EXIT

    # TODO 1: Acquire deployment lock with timeout
    #   Use acquire_deployment_lock with 300 second timeout
    #   If lock fails, exit with clear error message

    # TODO 2: Determine inactive environment
    #   Call get_inactive_environment()
    #   Validate it's different from active environment
    #   Log which environment will receive new deployment

    # TODO 3: Deploy to inactive environment with error handling
    #   Call deploy_to_environment "$inactive_env" "$version"
    #   If deployment fails, log error and exit without switching traffic
    #   Preserve deployment logs for debugging

    # TODO 4: Run health checks on deployed environment
    #   Call check_health_with_retry "$inactive_env" 3 5
    #   If health checks fail, trigger rollback and exit
    #   Include health check details in error message

    # TODO 5: Run smoke tests
    #   Call run_smoke_tests "$inactive_env" "$port"
    #   If smoke tests fail, trigger rollback and exit
    #   Capture and log specific test failures

    # TODO 6: Switch traffic with validation
    #   Call switch_traffic "$inactive_env"
    #   Validate switch succeeded by checking get_current_active_env()
    #   If switch fails, attempt emergency rollback to previous environment

    # TODO 7: Clean up previous environment (optional)
```

```
    #   Consider stopping old environment to save resources
    #   Only proceed if new environment is confirmed healthy post-switch

    log_error "deployment" "Deployment completed successfully: $version to $inactive_env"
}

deployment_failed() {
    local error_msg="$1"

    # TODO 1: Log comprehensive error context
    #   Include deployment version, environment, timestamp
    #   Capture recent logs from failing component

    # TODO 2: Determine if rollback is needed
    #   Check if traffic was already switched
    #   Check if failing environment is currently active

    # TODO 3: Send appropriate alert
    #   High severity if production impacted
    #   Include recovery actions taken

    # TODO 4: Release deployment lock
    #   Ensure lock is released even on failure

    # TODO 5: Exit with non-zero code
    #   Use different exit codes for different failure types
}

cleanup_temp_resources() {
    # TODO: Clean up any temporary files, containers, or resources
    #   created during deployment attempt
    true  # Placeholder
}

# Execute main if script is run directly
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    if [[ $# -lt 1 ]]; then
        echo "Usage: $0 <version>"
        exit 1
    fi
    source "$(dirname "$0")/error_handlers.sh"
    main "$1"
```

```
fi
```

Language-Specific Hints

Bash Error Handling Patterns:

- Use `set -eEuo pipefail` at script start for strict error handling
- Implement `trap` handlers for cleanup on exit (both success and failure)
- Use `local` variables within functions to avoid namespace pollution
- Always check return codes of critical commands: `if ! command; then handle_error; fi`
- For timeouts, use `timeout` command or `$SECONDS` variable with loop checking

State Management in Bash:

- Write state files atomically: `echo "value" > "${file}.tmp" && mv "${file}.tmp" "$file"`
- Include timestamps and metadata in state files for debugging
- Validate state file contents before using them
- Implement file locking for critical sections using `flock` command

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Deployment hangs indefinitely	Deployment lock held by stalled process	Check `/tmp/deployment.lock` contents and process status	Kill orphaned process; remove stale lock file
Health check passes but app errors	Superficial health check not testing business logic	Examine application logs; test actual endpoints	Enhance health check to validate critical functionality
Traffic switch appears successful but no traffic flows	Load balancer config syntax error or wrong upstream port	Check load balancer error logs; test upstream directly	Validate config with `nginx -t` before reload
Rollback loops repeatedly	False positive failure detection triggering rollback	Examine alert thresholds; check monitoring system health	Implement circuit breaker; increase failure thresholds
Database errors after migration	Backward compatibility violation	Query `schema_migrations` table; test old app version with new schema	Revert contract migration; ensure expand phase completed

Diagnostic Commands:

```bash
# Check deployment lock status
cat /tmp/deployment.lock 2>/dev/null || echo "No lock file"

# View last deployment errors
tail -100 /var/log/deployments.log | grep -A5 -B5 "ERROR"

# Test environment health directly
curl -f http://localhost:$BLUE_PORT/health
curl -f http://localhost:$GREEN_PORT/health

# Verify load balancer configuration
nginx -t 2>&1 | tail -20

# Check current active environment
cat $ACTIVE_ENV_FILE 2>/dev/null || echo "No active env file"

# List running containers by environment color
docker ps --filter "label=environment=blue" --format "table {{.ID}}\t{{.Names}}\t{{.Status}}"
```

Testing Strategy

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

A comprehensive testing strategy for blue-green deployment is like **quality control in an automobile assembly line**. You need to test each individual component (unit tests), verify they work together when assembled (integration tests), and finally perform stress tests to ensure the vehicle handles real-world road conditions safely (chaos tests). This systematic approach ensures that your deployment pipeline isn't just mechanically functional but also resilient to failures that inevitably occur in production environments.

Test Types and Scenarios

Mental Model: The Three-Layer Testing Pyramid for Blue-Green Deployments

Imagine testing a blue-green deployment system as a **three-layer pyramid of verification**:

1. **Foundation Layer (Unit Tests)** - Testing individual components in isolation. Each component (Environment Manager, Load Balancer Controller, etc.) is like a precision gear in a Swiss watch. You test each gear separately to ensure it rotates correctly, has the right number of teeth, and meshes properly with adjacent gears. This layer ensures each component fulfills its contract.
2. **Middle Layer (Integration Tests)** - Testing components working together. This is like assembling the watch movement and verifying that when you wind the mainspring, all gears turn in sync and the hands move. You're testing the interactions between components: does the Deployment Orchestrator correctly call the Environment Manager? Does the Load Balancer Controller properly validate health before switching?
3. **Apex Layer (Chaos Tests)** - Testing the system's resilience under failure. This is like deliberately introducing dust, magnetic fields, or physical shocks to the assembled watch and verifying it still keeps accurate time. You're testing the system's ability to handle real-world failures: network partitions, process crashes, disk full errors, and race conditions.

Unit Tests: Component-Level Verification

Unit tests verify each component in isolation, mocking dependencies. They should be fast, deterministic, and cover both happy paths and edge cases. Each component's interface (defined in previous sections) provides the contract to test against.

Environment Manager Unit Test Scenarios:

Test Scenario	Input	Expected Behavior	Validation Method
Start environment succeeds	`env_color="blue"`	Environment starts on correct port (8080), health endpoint becomes available	HTTP GET to `http://localhost:8080/health` returns 200 with `"environment": "blue"`
Start environment fails (port conflict)	`env_color="blue"` when port 8080 is occupied	Function returns error, clean error message logged	Exit code non-zero, error logged to stderr
Health check passes	Environment running normally	`check_health("blue")` returns `true`	Function returns exit code 0
Health check fails (process dead)	Environment process killed	`check_health("blue")` returns `false`	Function returns exit code 1
Get inactive environment	Active="blue"	`get_inactive_environment()` returns `"green"`	String comparison returns `"green"`
Environment identity injection	Environment variables set	Application receives correct `ENVIRONMENT_COLOR`, `PORT`, `DATABASE_URL`	Application logs/show environment variables match expected

Load Balancer Controller Unit Test Scenarios:

Test Scenario	Input	Expected Behavior	Validation Method
Config generation for environment	`env_color="green"`	Nginx config snippet with upstream pointing to port 8081	Generated config contains `server 127.0.0.1:8081`
Config validation (valid)	Valid Nginx config	`validate_load_balancer_config()` returns `true`	Exit code 0, no syntax errors
Config validation (invalid)	Malformed Nginx config	Function returns `false`, error message output	Exit code 1, error about syntax
Get current active environment	Active="blue" in state file	`get_current_active_env()` returns `"blue"`	String read from `ACTIVE_ENV_FILE` matches
Health validation passes	Green environment healthy	Traffic switch allowed	Function returns exit code 0
Health validation fails	Green environment unhealthy	Traffic switch blocked with error	Function returns exit code 1, error message about health check

Deployment Orchestrator Unit Test Scenarios:

Test Scenario	Input	Expected Behavior	Validation Method
Smoke tests pass	Healthy environment	`run_smoke_tests("green", 8081)` returns `true`	Exit code 0, all smoke test assertions pass
Smoke tests fail (critical endpoint down)	Environment missing `/api/health`	Function returns `false`, logs which test failed	Exit code 1, error about missing endpoint
Deployment to inactive environment	Version="v1.2.0", inactive="green"	`deploy_to_environment("green", "v1.2.0")` succeeds	Mock verification that deploy script called with correct arguments
Rollback triggered	Smoke test failure after deploy	`rollback_deployment("green")` called, traffic switched back to blue	Mock verification that `switch_traffic("blue")` called
Idempotent deployment	Same version deployed twice	Second deployment skips redundant steps or handles gracefully	Logs indicate "already deployed" or no-op

Database Migrator Unit Test Scenarios:

Test Scenario	Input	Expected Behavior	Validation Method
Migration application	`target_version="003", direction="up"`	Migration SQL files 001-003 applied, version table updated	Database schema matches expected, `schema_migrations` table has new rows
Migration idempotency	Same migration applied twice	Second application skips or safely handles already-applied migrations	Logs indicate "already applied", no duplicate rows in version table
Schema version read	Database with version "002" applied	`get_current_schema_version()` returns `"002"`	String comparison matches
Migration compatibility check	Blue=v1.1.0, Green=v1.2.0, Schema=v2	`validate_migration_compatibility()` returns `true` if schema compatible with both	Boolean true if expand phase completed, false if contract phase would break old version
Migration rollback	`target_version="001", direction="down"`	Migration 002-003 rolled back, schema reverted	Database schema matches version 001, version table updated

Integration Tests: Pipeline-Level Verification

Integration tests verify that components work together correctly. These tests require actual infrastructure (processes, network) and are slower but more realistic. They test the complete deployment pipeline from code commit to traffic switch.

Key Integration Test Scenarios:

1. **Complete Deployment Flow Test**
   - **Setup**: Blue environment active (v1.0.0), green environment inactive (stopped)
   - **Action**: Run deployment script with new version v1.1.0
   - **Verification**:
     1. Green environment starts with v1.1.0 on port 8081
     2. Smoke tests pass on green environment
     3. Traffic switches from blue to green
     4. Load balancer now routes requests to port 8081

5. Blue environment remains running (for rollback capability)

2. **Rollback Flow Test**
   - **Setup**: Blue active (v1.0.0), green just deployed (v1.1.0) but smoke tests failing
   - **Action**: Deployment script detects smoke test failure
   - **Verification**:
     1. Traffic NOT switched to green
     2. Script triggers rollback (or exits with error for manual rollback)
     3. Traffic remains on blue environment
     4. Green environment stopped or marked as failed
     5. Alert triggered about deployment failure
3. **Database Migration Compatibility Test**
   - **Setup**: Blue running v1.1.0 (expects schema v2), database at schema v2
   - **Action**: Apply forward migration to schema v3 (expand phase only)
   - **Verification**:
     1. Migration applies successfully
     2. Blue environment (v1.1.0) continues working with schema v3 (backward compatibility)
     3. Green environment with v1.2.0 (uses new schema features) can be deployed
     4. After traffic switched to green, contract phase migration can be applied
4. **Concurrent Request Handling During Switch**
   - **Setup**: Blue active, high load simulated with ongoing requests
   - **Action**: Trigger traffic switch to green
   - **Verification**:
     1. In-flight requests to blue complete without interruption
     2. New requests go to green after switch
     3. No 5xx errors during switch window
     4. Connection draining works correctly

Chaos Tests: Resilience Verification

Chaos tests deliberately introduce failures to verify the system's resilience. These tests ensure that when (not if) things go wrong, the system fails safely and provides clear signals for recovery.

Chaos Test Scenarios:

Failure Type	Injection Method	Expected System Response	Recovery Verification
Environment process crash during deployment	Kill application process on inactive environment after deploy but before switch	Deployment orchestrator detects failed health check, blocks traffic switch, triggers rollback	System remains on previous environment, failed environment marked unhealthy
Database connection failure	Block database access for green environment after deployment	Smoke tests fail (database connectivity check), rollback triggered	Traffic stays on blue, alerts fire for database connectivity
Load balancer config corruption	Manually corrupt Nginx config file before switch	`validate_load_balancer_config()` fails, switch aborted, original config preserved	Traffic continues flowing to active environment, error logged
Network partition between load balancer and environment	Use firewall rules to block traffic to green environment's port	Health checks fail, traffic switch prevented even if application running	System detects unreachable environment, prevents cutover
Disk full on deployment server	Fill disk before artifact deployment	Deployment fails with clear disk space error, no partial deployment	System alerts on disk space, manual cleanup required
Race condition: simultaneous deployment attempts	Run two deployment scripts concurrently	First deployment locks deployment process, second fails or waits	Only one deployment proceeds at a time, second fails with "deployment in progress"
Partial migration failure	Apply migration that fails midway (syntax error)	Migration transaction rolled back, schema unchanged, deployment fails	Database remains at previous consistent state, deployment aborted

Architecture Decision: Testing Strategy Approach

> **Decision: Hybrid Testing Approach with Emphasis on Integration Tests**
>
> **Context**: We need to verify the blue-green deployment system works correctly both at component level and as an integrated pipeline. The system involves multiple moving parts (environments, load balancer, database) that must coordinate precisely. Traditional unit-test-only approaches miss integration failures, while full end-to-end tests are slow and flaky.
>
> **Options Considered**:
>
> 1. **Unit Tests Only**: Fast, reliable tests that mock all dependencies. Components are tested in isolation.
> 2. **Integration Tests Only**: Real infrastructure tests that verify the complete pipeline works end-to-end.
> 3. **Hybrid Approach**: Comprehensive unit tests for components plus critical integration tests for key workflows.
>
> **Decision**: Hybrid Approach (Option 3) with approximately 70% unit tests, 20% integration tests, and 10% chaos tests.
>
> **Rationale**:
>
> - Unit tests provide rapid feedback during development and catch logic errors early.
> - Integration tests catch configuration and interaction problems that unit tests miss (e.g., port conflicts, network timeouts).
> - Chaos tests validate resilience requirements that are critical for zero-downtime deployments.
> - The 70/20/10 ratio balances speed with confidence: most testing is fast (unit), but critical paths get realistic validation.
> - This approach aligns with the testing pyramid model, optimizing for both developer velocity and production reliability.
>
> **Consequences**:
>
> - **Positive**: Catches both component bugs and integration issues. Provides confidence in production readiness.
> - **Negative**: Requires more test infrastructure (Docker containers, test databases). Integration tests are slower to run.
> - **Mitigation**: Run unit tests on every commit, integration tests on pull requests, chaos tests nightly.

Option	Pros	Cons	Why Not Chosen
Unit Tests Only	Fast, reliable, easy to debug	Misses integration issues, false confidence	Production failures occur despite high unit test coverage
Integration Tests Only	Realistic, catches configuration issues	Slow, flaky, hard to debug failures	Too slow for development cycle, difficult to maintain
Hybrid Approach	Best of both worlds, balances speed and realism	More complex test suite, needs infrastructure	**CHOSEN** - Worth the complexity for production reliability

Common Pitfalls in Testing Blue-Green Deployments

⚠️ Pitfall: Testing Only the Happy Path

- **Description**: Only testing successful deployments where everything works perfectly.
- **Why It's Wrong**: Production is where things go wrong. If you haven't tested failures, your system will fail unpredictably.
- **How to Fix**: Dedicate at least 30% of test scenarios to failure cases: failed health checks, network timeouts, disk full, race conditions.

⚠️ Pitfall: Mocking Too Much in Unit Tests

- **Description**: Mocking away all external dependencies (network, filesystem, processes) to the point that tests don't verify real behavior.
- **Why It's Wrong**: Tests pass but the real component fails because mocked behavior doesn't match reality (e.g., assuming command always succeeds).
- **How to Fix**: Use realistic mocks that simulate real failures. Consider using integration tests for critical external interactions.

⚠️ Pitfall: Not Testing Rollback Under Load

- **Description**: Testing rollback with no traffic, then discovering in production that rollback under load causes cascading failures.
- **Why It's Wrong**: Rollback mechanisms must work when the system is stressed, not just when idle.
- **How to Fix**: Run rollback tests with simulated load (using tools like `siege` or `wrk`) to verify performance and connection draining.

⚠️ Pitfall: Assuming Database State is Always Clean

- **Description**: Tests start with empty database each time, not testing migrations on production-sized data.
- **Why It's Wrong**: Migrations that work on empty test databases may timeout or deadlock on large production tables.
- **How to Fix**: Create test databases with representative data volumes (thousands of rows) and test migrations against them.

⚠️ Pitfall: Not Testing Configuration Variants

- **Description**: Testing only one configuration (e.g., default ports) and missing issues with different settings.

- **Why It's Wrong**: In production, environments may have different resource limits, network topologies, or security constraints.
- **How to Fix**: Test with multiple configuration profiles: different ports, memory limits, database connection strings.

Milestone Checkpoints

Each milestone has specific acceptance criteria. These checkpoints provide concrete verification steps to ensure you've implemented each milestone correctly.

Milestone 1: Dual Environment Setup Checkpoints

Goal: Verify two identical environments can run simultaneously with proper isolation.

Checkpoint	Command to Run	Expected Output	What to Verify
Both environments start independently	`start_environment "blue" && start_environment "green"`	Both commands succeed (exit code 0)	Processes listening on ports 8080 and 8081
Environment identity via health endpoint	`curl -s http://localhost:8080/health	jq .environment`	`"blue"`
	`curl -s http://localhost:8081/health	jq .environment`	`"green"`
Environment isolation (deploy to one)	Deploy new version to blue only, then check green	Green health endpoint shows old version, blue shows new version	Versions differ, proving environments are isolated
Configuration injection	Check environment variables in application logs	Each environment shows its specific `PORT`, `DATABASE_URL`, `ENVIRONMENT_COLOR`	Configuration properly injected per environment
Concurrent operation	Send requests to both environments simultaneously	Both respond successfully without interference	Environments don't share resources (ports, temp files)

Manual Verification Steps:

1. Start both environments: `./scripts/start_environment.sh blue && ./scripts/start_environment.sh green`
2. Verify both are running: `ps aux | grep "app.*--port"` should show two processes with different ports
3. Test health endpoints:

```bash
curl -f http://localhost:8080/health  # Should return 200
curl -f http://localhost:8081/health  # Should return 200
```

4. Check environment identity in responses matches expected color
5. Deploy a version change to blue only and verify green remains unchanged

Milestone 2: Load Balancer Switching Checkpoints

Goal: Verify traffic can be switched between environments atomically without dropping connections.

Checkpoint	Command to Run	Expected Output	What to Verify
Load balancer routes to active environment	`curl -s http://localhost:80/health	jq .environment`	Matches active environment in `ACTIVE_ENV_FILE`
Traffic switch execution	`switch_traffic "green"`	Exit code 0, "Traffic switched to green" message	Command succeeds
Post-switch verification	`curl -s http://localhost:80/health	jq .environment`	`"green"`
Graceful reload (no connection drop)	While running load test, switch traffic	No 5xx errors in load test logs	In-flight requests complete, new requests go to new environment
Health check validation before switch	Disable green health endpoint, then try switch	Exit code 1, "Health check failed" error	Switch prevented when target unhealthy
Rollback capability	`switch_traffic "blue"`	Exit code 0, traffic returns to blue	Instant reversion possible
Config validation	`validate_load_balancer_config`	Exit code 0 for valid config	Syntax checking works

Manual Verification Steps:

1. Start with blue active: `echo "blue" > /tmp/active_environment`
2. Start load balancer: `sudo nginx -c /path/to/nginx.conf`
3. Verify initial routing: `curl http://localhost/health` should return blue environment

4. Start a continuous request loop in another terminal: `while true; do curl -s http://localhost/health | grep environment; sleep 0.1; done`
5. Switch traffic: `./scripts/switch_traffic.sh green`
6. Observe request loop: Should show transition from "blue" to "green" with no gaps
7. Verify nginx reloaded gracefully: `sudo nginx -t` should pass, `ps aux | grep nginx` should show old worker processes finishing
8. Test rollback: `./scripts/switch_traffic.sh blue` and verify traffic returns

Milestone 3: Deployment Automation Checkpoints

Goal: Verify the full deployment pipeline automates build, deploy, test, and switch steps.

Checkpoint	Command to Run	Expected Output	What to Verify
Full deployment script	`./deploy.sh v1.2.0`	Script runs through all stages, ends with "Deployment complete"	Complete automation from build to traffic switch
Deployment to inactive environment	Monitor during deployment	Active environment continues serving traffic uninterrupted	Zero downtime maintained
Smoke test execution	Script output shows "Running smoke tests..."	"Smoke tests passed" message	Critical functionality validated
Smoke test failure handling	Break a critical endpoint in new version	Script exits with error before traffic switch	Switch blocked when smoke tests fail
Post-deployment verification	After deployment, check active environment	New version running and receiving traffic	Deployment successful end-to-end
Idempotent deployment	Run same deployment twice	Second run skips or handles already-deployed version gracefully	No duplicate work or errors

Manual Verification Steps:

1. Start with blue active (v1.0.0), green stopped
2. Run deployment: `./deploy.sh v1.1.0`
3. Observe script output:
   - Builds artifact
   - Deploys to green (inactive environment)
   - Starts green with v1.1.0
   - Runs smoke tests against green
   - Switches traffic to green
   - Reports success
4. Verify result: `curl http://localhost/health` shows green environment with v1.1.0
5. Verify blue still running (for rollback): `curl http://localhost:8080/health` shows blue with v1.0.0
6. Test smoke test failure:
   - Modify smoke tests to expect wrong response
   - Run deployment again with v1.2.0
   - Verify script fails before traffic switch
   - Verify traffic remains on green (v1.1.0)

Milestone 4: Rollback & Database Migrations Checkpoints

Goal: Verify safe rollback and backward-compatible database migrations.

Checkpoint	Command to Run	Expected Output	What to Verify
Instant rollback on failure	Break smoke test, run deployment	Script fails, triggers automatic rollback	Traffic reverted to previous environment within seconds
Manual rollback command	`rollback_deployment "green"`	Exit code 0, traffic switches back to blue	Manual intervention possible
Expand-contract migration	Apply expand-phase migration	Database schema gains nullable column, both app versions work	Backward compatibility maintained
Dual version compatibility	Blue (old) and green (new) both running	Both can read/write database without errors	Expand phase successful
Contract phase application	After traffic switched to new version, apply contract migration	Old columns removed, constraints added	Schema optimized after old version retired
Migration idempotency	Run same migration twice	Second run reports "already applied" or no-op	Safe for retry
Migration rollback	Apply down migration	Schema reverts to previous version	Emergency rollback possible

Manual Verification Steps:

1. Set up database with initial schema
2. Deploy blue with v1.0.0 (works with current schema)
3. Apply expand migration (add nullable column): `./migrate.sh up 001_add_column.sql`
4. Verify blue still works with new schema
5. Deploy green with v1.1.0 (uses new column): `./deploy.sh v1.1.0`
6. Verify green works with new schema
7. Switch traffic to green
8. After confirming green works, apply contract migration (make column NOT NULL, drop old column)
9. Test rollback scenario:
    - Deploy v1.2.0 to blue (now inactive)
    - Make it fail smoke tests
    - Verify automatic rollback keeps traffic on green
    - Verify database hasn't been corrupted

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Unit Testing Framework	Shell scripts with `test` command and assertions	`bats` (Bash Automated Testing System) for structured tests
Integration Test Runner	Custom Bash scripts that start infrastructure	Docker Compose for environment isolation, `pytest` with fixtures
Chaos Testing	Manual failure injection	`chaos-toolkit` for automated chaos experiments
Load Testing	`siege` or `ab` (Apache Bench) for simple tests	`wrk` or `k6` for advanced scripting and metrics
Database Testing	Direct `psql` / `mysql` commands	Test containers with predefined datasets
Mocking	Simple shell functions that echo expected output	`mock` command or custom wrapper scripts

Recommended File Structure for Tests

```
project-root/
  scripts/                    # Main deployment scripts
    deploy.sh
    switch_traffic.sh
    rollback.sh
    ...

  tests/                      # All test files
    unit/                     # Unit tests
      test_environment_manager.sh
      test_load_balancer_controller.sh
      test_deployment_orchestrator.sh
      test_database_migrator.sh
      helpers/                # Test utilities
        assert.sh             # Assertion functions
        mock_helpers.sh       # Mock creation helpers

    integration/              # Integration tests
      test_full_deployment.sh
      test_rollback.sh
      test_migration_compatibility.sh
      fixtures/               # Test data and configs
        nginx-test.conf
        sample-app/

    chaos/                    # Chaos tests
      test_process_crash.sh
      test_network_partition.sh
      test_disk_full.sh

    load/                     # Load tests
      test_switch_under_load.sh

    run_all_tests.sh          # Test runner script

  src/                        # Application code
    app.py                    # Sample application
    health_handler.py         # Health endpoint logic

  migrations/                 # Database migrations
    001_initial_schema.sql
    002_add_feature_flag.sql
    ...

  configs/                    # Configuration files
    nginx.conf.template
    environment_vars.blue
    environment_vars.green
```

Infrastructure Starter Code: Test Assertion Helpers

File: `tests/unit/helpers/assert.sh`

```bash
#!/bin/bash

# Test assertion helpers for Bash testing


# Colors for output

RED='\033[0;31m'

GREEN='\033[0;32m'

YELLOW='\033[1;33m'

NC='\033[0m' # No Color


# Assert that command succeeds (exit code 0)

assert_success() {

    local cmd="$*"

    eval "$cmd" > /dev/null 2>&1

    local exit_code=$?



    if [ $exit_code -eq 0 ]; then

        echo -e "${GREEN}√ assert_success: '$cmd'${NC}"

        return 0

    else

        echo -e "${RED}╳ assert_success: '$cmd' failed with exit code $exit_code${NC}"

        exit 1

    fi

}


# Assert that command fails (exit code non-zero)

assert_failure() {

    local cmd="$*"

    eval "$cmd" > /dev/null 2>&1

    local exit_code=$?



    if [ $exit_code -ne 0 ]; then

        echo -e "${GREEN}√ assert_failure: '$cmd' failed as expected${NC}"

        return 0

    else

        echo -e "${RED}╳ assert_failure: '$cmd' succeeded but should have failed${NC}"

        exit 1

    fi

}


# Assert two strings are equal

assert_equal() {

    local expected="$1"
```

```
    local actual="$2"
    local message="${3:-}"

    if [ "$expected" = "$actual" ]; then
        echo -e "${GREEN}✓ assert_equal: '$expected' == '$actual' $message${NC}"
        return 0
    else
        echo -e "${RED}✗ assert_equal: expected '$expected', got '$actual' $message${NC}"
        exit 1
    fi
}

# Assert JSON field value matches expected
assert_json_field() {
    local json="$1"
    local field="$2"
    local expected="$3"
    local actual=$(echo "$json" | jq -r ".$field")

    assert_equal "$expected" "$actual" "for field '$field'"
}

# Assert HTTP response code
assert_http_code() {
    local url="$1"
    local expected_code="$2"
    local actual_code=$(curl -s -o /dev/null -w "%{http_code}" "$url")

    assert_equal "$expected_code" "$actual_code" "for URL $url"
}

# Assert process is running
assert_process_running() {
    local process_name="$1"
    local count=$(pgrep -f "$process_name" | wc -l)

    if [ $count -gt 0 ]; then
        echo -e "${GREEN}✓ assert_process_running: '$process_name' is running${NC}"
        return 0
    else
        echo -e "${RED}✗ assert_process_running: '$process_name' is not running${NC}"
        exit 1
```

```
    fi
}

# Assert port is listening
assert_port_listening() {
    local port="$1"

    if nc -z localhost "$port" 2>/dev/null; then
        echo -e "${GREEN}√ assert_port_listening: port $port is listening${NC}"
        return 0
    else
        echo -e "${RED}╳ assert_port_listening: port $port is not listening${NC}"
        exit 1
    fi
}

# Setup and teardown helpers
setup_test() {
    echo -e "${YELLOW}Setting up test...${NC}"
    # Kill any existing test processes
    pkill -f "test_app.*--port" 2>/dev/null || true
    # Clean up test files
    rm -f /tmp/test_*.log /tmp/test_*.pid
}

teardown_test() {
    echo -e "${YELLOW}Tearing down test...${NC}"
    # Clean up processes
    pkill -f "test_app.*--port" 2>/dev/null || true
    # Remove test files
    rm -f /tmp/test_*.log /tmp/test_*.pid
}
```

Core Logic Skeleton: Unit Test for Environment Manager

File: `tests/unit/test_environment_manager.sh`

BASH

```bash
#!/bin/bash

# Unit tests for Environment Manager component


source "$(dirname "$0")/helpers/assert.sh"


# Mock functions for dependencies

mock_curl() {

    if [ "$1" = "http://localhost:8080/health" ]; then

        echo '{"status":"healthy","environment":"blue","version":"v1.0.0","uptime_seconds":100}'

    elif [ "$1" = "http://localhost:8081/health" ]; then

        echo '{"status":"healthy","environment":"green","version":"v1.0.0","uptime_seconds":50}'

    else

        return 1

    fi

}


# Mock the actual curl command

alias curl=mock_curl


# TODO 1: Test start_environment function

# - Mock the application startup command

# - Verify correct port is used based on environment color

# - Verify environment variables are set

test_start_environment() {

    echo "Testing start_environment..."



    # TODO: Implement test

    # 1. Create a mock application script that echoes its arguments

    # 2. Call start_environment "blue"

    # 3. Verify mock app was called with --port 8080

    # 4. Verify ENVIRONMENT_COLOR=blue was passed

    echo "  TODO: Implement start_environment test"

}


# TODO 2: Test check_health function

# - Mock HTTP response for health endpoint

# - Test healthy case returns true (exit code 0)

# - Test unhealthy case returns false (exit code 1)

test_check_health() {

    echo "Testing check_health..."



    # TODO: Implement test

    # 1. Mock curl to return healthy JSON for blue
```

```
    # 2. Call check_health "blue"
    # 3. Assert exit code is 0
    # 4. Mock curl to return error or unhealthy JSON
    # 5. Call check_health "blue"
    # 6. Assert exit code is 1
    echo "  TODO: Implement check_health test"
}

# TODO 3: Test get_inactive_environment function
# - Mock ACTIVE_ENV_FILE with "blue"
# - Verify function returns "green"
# - Mock ACTIVE_ENV_FILE with "green"
# - Verify function returns "blue"
test_get_inactive_environment() {
    echo "Testing get_inactive_environment..."

    # TODO: Implement test
    # 1. Create temporary ACTIVE_ENV_FILE with "blue"
    # 2. Call get_inactive_environment
    # 3. Assert output is "green"
    # 4. Update file to "green"
    # 5. Call get_inactive_environment
    # 6. Assert output is "blue"
    echo "  TODO: Implement get_inactive_environment test"
}

# TODO 4: Test environment isolation
# - Start blue environment
# - Start green environment
# - Verify they use different ports
# - Verify they can run concurrently
test_environment_isolation() {
    echo "Testing environment isolation..."

    # TODO: Implement test
    # 1. Start blue environment (mock)
    # 2. Start green environment (mock)
    # 3. Verify both processes are running
    # 4. Verify they're listening on different ports
    # 5. Send request to each, verify correct identity in response
    echo "  TODO: Implement environment isolation test"
```

```
}

# Main test runner
main() {
    echo "Running Environment Manager unit tests..."
    setup_test

    test_start_environment
    test_check_health
    test_get_inactive_environment
    test_environment_isolation

    teardown_test
    echo -e "${GREEN}All tests passed!${NC}"
}

# Run tests if script is executed directly
if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    main
fi
```

Integration Test Skeleton: Full Deployment Flow

File: `tests/integration/test_full_deployment.sh`

BASH

```bash
#!/bin/bash

# Integration test for full deployment flow

source "$(dirname "$0")/../unit/helpers/assert.sh"

# TODO 1: Setup test infrastructure
# - Start with clean state: no environments running
# - Initialize database with test schema
# - Create test application artifact
setup_integration_test() {
    echo "Setting up integration test..."

    # TODO: Implement
    # 1. Stop any running blue/green environments
    # 2. Clear ACTIVE_ENV_FILE
    # 3. Initialize test database with schema
    # 4. Build test application artifact (v1.0.0)
    # 5. Deploy v1.0.0 to blue, mark as active
    echo "  TODO: Implement integration test setup"
}

# TODO 2: Test complete deployment flow
# - Run deployment script for v1.1.0
# - Verify all steps execute correctly
# - Verify traffic switches after successful smoke tests
test_deployment_flow() {
    echo "Testing complete deployment flow..."

    # TODO: Implement
    # 1. Verify blue is initially active with v1.0.0
    # 2. Run deploy.sh v1.1.0
    # 3. Verify script executes steps in order:
    #    a. Builds v1.1.0 artifact
    #    b. Deploys to green (inactive environment)
    #    c. Starts green with v1.1.0
    #    d. Runs smoke tests against green
    #    e. Switches traffic to green
    # 4. Verify green is now active with v1.1.0
    # 5. Verify blue still running with v1.0.0 (for rollback)
    echo "  TODO: Implement deployment flow test"
}
```

```
# TODO 3: Test deployment with failing smoke tests
# - Make smoke tests fail for new version
# - Verify deployment stops before traffic switch
# - Verify rollback is triggered or error reported
test_deployment_with_failing_smoke_tests() {
    echo "Testing deployment with failing smoke tests..."

    # TODO: Implement
    # 1. Setup: blue active v1.0.0
    # 2. Create v1.1.0 that will fail smoke tests
    # 3. Run deploy.sh v1.1.0
    # 4. Verify:
    #    a. Deployment to green happens
    #    b. Smoke tests fail
    #    c. Traffic NOT switched
    #    d. Script exits with error
    #    e. Traffic remains on blue
    # 5. Clean up green environment
    echo "  TODO: Implement failing smoke test deployment test"
}

# TODO 4: Test rollback mechanism
# - Deploy bad version that passes smoke tests but has hidden bug
# - Simulate bug discovery
# - Trigger rollback
# - Verify traffic reverts to previous version
test_rollback_mechanism() {
    echo "Testing rollback mechanism..."

    # TODO: Implement
    # 1. Deploy v1.1.0 successfully (green active)
    # 2. Discover bug in v1.1.0
    # 3. Run rollback_deployment "green"
    # 4. Verify:
    #    a. Traffic switches back to blue (v1.0.0)
    #    b. Green is stopped or marked unhealthy
    #    c. Rollback completes within seconds
    echo "  TODO: Implement rollback test"
}

main() {
```

```
    echo "Running integration tests for full deployment flow..."


    setup_integration_test
    test_deployment_flow
    test_deployment_with_failing_smoke_tests
    test_rollback_mechanism


    echo -e "${GREEN}Integration tests completed!${NC}"
}

if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    main
fi
```

Milestone Checkpoint Commands

File: `tests/run_milestone_checkpoints.sh`

```bash
#!/bin/bash

# Run milestone verification checkpoints

echo "=== Milestone 1: Dual Environment Setup ==="

echo "1. Starting both environments..."

./scripts/start_environment.sh blue &

./scripts/start_environment.sh green &

sleep 2

echo "2. Verifying both environments are running..."

curl -s http://localhost:8080/health | jq -r '.environment' | grep -q "blue" && echo "√ Blue environment running"

curl -s http://localhost:8081/health | jq -r '.environment' | grep -q "green" && echo "√ Green environment running"

echo "3. Testing environment isolation..."

BLUE_VERSION=$(curl -s http://localhost:8080/health | jq -r '.version')

GREEN_VERSION=$(curl -s http://localhost:8081/health | jq -r '.version')

echo "  Blue version: $BLUE_VERSION"

echo "  Green version: $GREEN_VERSION"

[ "$BLUE_VERSION" = "$GREEN_VERSION" ] && echo "√ Environments have same version (isolated deployment not tested yet)"

echo ""

echo "=== Milestone 2: Load Balancer Switching ==="

echo "1. Setting blue as active..."

echo "blue" > /tmp/active_environment

./scripts/switch_traffic.sh blue

echo "2. Verifying load balancer routes to blue..."

ACTIVE_ENV=$(curl -s http://localhost/health 2>/dev/null | jq -r '.environment' || echo "none")

[ "$ACTIVE_ENV" = "blue" ] && echo "√ Load balancer routing to blue"

echo "3. Switching traffic to green..."

./scripts/switch_traffic.sh green

sleep 1

echo "4. Verifying load balancer now routes to green..."

NEW_ACTIVE_ENV=$(curl -s http://localhost/health 2>/dev/null | jq -r '.environment' || echo "none")

[ "$NEW_ACTIVE_ENV" = "green" ] && echo "√ Load balancer switched to green"

echo ""

echo "=== Milestone 3: Deployment Automation ==="

echo "1. Running deployment script (dry run)..."

# TODO: Run actual deployment script with test version

echo "  TODO: Implement actual deployment test"

echo ""
```

```
echo "=== Milestone 4: Rollback & Database Migrations ==="

echo "1. Testing database migration..."

# TODO: Run migration and verify

echo "  TODO: Implement migration test"


echo "2. Testing rollback..."

# TODO: Trigger rollback and verify

echo "  TODO: Implement rollback test"


echo ""

echo "=== Summary ==="

echo "Milestone 1: Dual Environment Setup - $( [ -n "$BLUE_VERSION" ] && echo "PASS" || echo "FAIL" )"

echo "Milestone 2: Load Balancer Switching - $( [ "$NEW_ACTIVE_ENV" = "green" ] && echo "PASS" || echo "FAIL" )"

echo "Milestone 3: Deployment Automation - TODO"

echo "Milestone 4: Rollback & Database Migrations - TODO"
```

Language-Specific Hints: Bash Testing

1. **Use `set -e` and `set -u`**: Start test scripts with `set -euo pipefail` to catch errors early.
2. **Mock commands with functions**: Override system commands in tests using shell functions: `curl() { echo '{"status":"healthy"}'; }`
3. **Temporary files**: Use `mktemp` for test files: `TEST_FILE=$(mktemp /tmp/test.XXXXXX)`
4. **Cleanup with traps**: Set up cleanup on exit: `trap 'rm -f $TEST_FILE' EXIT`
5. **Test exit codes**: Check command success with `if command; then ...` or `command && echo "success" || echo "failure"`
6. **JSON parsing**: Use `jq` for reliable JSON parsing in tests: `version=$(echo $json | jq -r '.version')`
7. **Parallel test isolation**: Use different ports and temp file names for each test to avoid conflicts.
8. **Timeout handling**: Use `timeout` command to prevent hanging tests: `timeout 5s curl http://localhost:8080/health`

Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Environment differences, timing issues	Compare environment variables, check for hardcoded paths	Use `env` in CI script, add wait/retry loops for startup
Health check passes but app errors	Health endpoint doesn't test actual functionality	Extend health check to test critical business logic	Add deeper health checks that exercise database, external services
Traffic switch test flakes	Race condition with connection draining	Check nginx logs for active connections during reload	Add sleep before checking post-switch, verify old workers exit
Migration test fails intermittently	Database not cleaned between tests	Check if test data persists between runs	Add proper teardown that drops/recreates test database
Smoke tests timeout	Application startup too slow	Check application logs for initialization delays	Increase timeout in smoke tests, optimize app startup
Rollback doesn't revert traffic	`ACTIVE_ENV_FILE` not updated correctly	Verify file permissions, check for race conditions	Use atomic file writes, add verification after switch

Debugging Guide

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

Even with a meticulously designed system like our blue-green deployment pipeline, things will go wrong. The complexity of managing two parallel environments, a traffic-switching load balancer, automated deployments, and backward-compatible database migrations creates numerous potential failure points. This debugging

guide serves as your **field manual** for diagnosing and resolving issues when they arise. Think of yourself as a **system detective**—you'll follow clues, examine evidence, and reconstruct events to understand what broke and why.

The key mental model for debugging this system is the **Request Journey**. A user request travels through a chain of components: from the load balancer, to the active environment (blue or green), to the application, and finally to the database. A failure can occur at any point along this journey. Your job is to trace the request's path, identify where it deviates from the expected behavior, and understand which component's state or configuration caused the deviation. This guide provides the tools and techniques to perform that tracing effectively.

Common Bugs: Symptom → Cause → Fix

This table catalogs the most frequent issues encountered during development and operation of a blue-green deployment system. For each symptom, we identify the probable root cause, provide concrete steps to diagnose the issue, and prescribe a fix.

Symptom	Root Cause	How to Diagnose	Fix
Health check passes but app returns errors to users.	The health check endpoint (`/health`) is overly simplistic and returns `200 OK` even when critical downstream dependencies (database, cache, external APIs) are failing. The application process is running, but it's in a degraded state.	1. Manually call the health endpoint: `curl http://<env-host>:<port>/health` . Verify the `database_connected` field and other deep health indicators. 2. Check application logs for the specific environment for connection errors or timeouts. 3. Test a non-health endpoint (e.g., a core API like `/api/data`) to see if it fails.	Enhance the `check_` the `/health` endpo **health checks**. Verify connections and interr `BlueGreenHandler` `"status":"health` subsystems are opera
Traffic does not switch after running `switch_traffic(target_env)` .	The `ACTIVE_ENV_FILE` was not updated, the load balancer configuration was generated incorrectly, or the graceful reload command (e.g., `nginx -s reload`) failed silently. The load balancer is still using its old configuration.	1. Check the contents of `ACTIVE_ENV_FILE` : `cat /tmp/active_environment` . Does it match the intended target? 2. Examine the generated load balancer config file (e.g., `/etc/nginx/conf.d/bluegreen.conf`). Does the `upstream active_backend` point to the correct environment's port? 3. Check load balancer error logs: `sudo tail -f /var/log/nginx/error.log` . Look for syntax errors from the last reload. 4. Verify the reload command succeeded by checking its exit code in your script or looking for Nginx worker process IDs (they should change after a reload).	Ensure `switch_tra` **validated**. The functio config for the target er `validate_load_ba` 3. Update `ACTIVE_E` Execute the graceful r *return code*. If any ste should abort and not u
Database errors appear immediately after a traffic switch to the new environment.	The new application version is not **backward compatible** with the current database schema. The `expand` phase of a migration may be missing, or the new code expects a column or constraint that doesn't exist yet.	1. Identify the error in the application logs of the newly active environment. It will likely be a SQL error about a missing column or table. 2. Run `get_current_schema_version()` and compare it to the schema version the application expects (often embedded in migration files). 3. Use `validate_migration_compatibility(app_version_blue, app_version_green)` to check if the current schema supports both versions. It will likely fail.	**Rollback immediatel** `rollback_deploym` to switch traffic back t Then, fix the database ensuring you follow th **pattern**. Deploy a mig new column as `NULL` deploy the new code t both the old and new s run the contract migra column `NOT NULL`) deployment.
The deployment script hangs indefinitely during the `run_smoke_tests` phase.	The smoke tests are making requests to an incorrect URL or port, the environment is not fully healthy yet, or a test has an infinite loop or long timeout. The orchestrator is waiting for a test pass/fail signal that never arrives.	1. Check the deployment log (`DEPLOYMENT_LOG`) to see the last step executed. 2. Manually run the smoke test command that the script uses, but against the inactive environment's direct port (e.g., `curl http://localhost:8081/api/smoke`). Observe if it hangs or returns. 3. Verify the inactive environment is actually listening on its port: `sudo netstat -tlnp	grep `. 4. Check if the smoke timeouts configured.
After a rollback, users still experience errors or see the new (buggy) version.	**Connection draining** was not properly implemented. When traffic was switched back, existing persistent connections (like WebSockets or long-lived HTTP/1.1 keep-alive connections) to the failed new environment were not gracefully closed and continue to be served by the old process.	1. Check the load balancer configuration for `graceful` or `drain` settings. Nginx's `upstream` block should have `server` directives with a `drain` parameter or similar. 2. Monitor active connections to the blue and green environment ports *after* a switch. You may see a decreasing number on the old active environment if draining is working. 3. Reproduce with a long-lived `curl` request during a switch and see if it gets cut off.	Configure the load bal connection draining. F `ngx_http_upstrea` mark a backend as `d` `switch_traffic` p old active backend as short period (e.g., 30 s the configuration to re allows in-flight reques
Deployment fails because the "inactive" environment is already running a different version or is unhealthy.	The `get_inactive_environment()` logic is flawed, or a previous deployment failed partially, leaving the environment in an inconsistent state. The orchestrator assumes it can deploy to the inactive side, but that assumption is false.	1. Run `get_inactive_environment()` manually and verify its output. 2. Check the health of both environments using `check_health("blue")` and `check_health("green")` . 3. Inspect the version of the application running on each environment (via their `/health` endpoint).	Make `get_inactiv` **idempotent and state** just assume the non-a free. It must verify the stopped or in a clean s deployment. If not, the either attempt to clear (`stop_environmen` deployment with a clea
Database migration locks the table, causing application timeouts during deployment.	The migration script uses an `ALTER TABLE ... ADD COLUMN` with a long-running operation (like adding a	1. Observe database monitoring for long-running queries and locks during the migration phase. 2. Check the specific SQL statements in your migration files. Look	Use **safe migration te** adding columns to larg using tools like `pt-o`

Symptom	Root Cause	How to Diagnose	Fix
	column with a default value to a huge table) that locks the table, blocking both old and new application versions.	for operations that require exclusive table locks. 3. Application logs will show an increase in database timeout errors around the time the migration ran.	`change` (for MySQL) migration in batches w exclusive lock. For Po `ALTER TABLE ...` *without* a default value metadata-only operati data in batches later.
The load balancer returns 502 Bad Gateway after a traffic switch.	The newly active environment's application process crashed or is not listening on the expected port. The load balancer's health check may not have caught a transient failure, or the application failed immediately after the health check passed.	1. Check the load balancer's error log for "connect() failed" or "connection refused" messages pointing to the new environment's port. 2. Verify the application process is running on the target environment: `ps aux	grep . 3. Chec application boun port (e.g., BLUE_PORT or GRE might be binding 0.0.0.0:8080 when t expects localhost:80
Two deployments are triggered almost simultaneously, causing race conditions.	Lack of a **deployment lock**. While one deployment is in progress (e.g., building), another CI/CD pipeline run starts, leading to two processes trying to manage environments and switch traffic concurrently.	1. Check deployment logs for overlapping timestamps from two different pipeline executions. 2. Observe inconsistent state: the load balancer might point to blue, but the `ACTIVE_ENV_FILE` says green, or two different versions might be deployed to the same environment.	Implement a **distribut** the deployment orches lock could be a file (e. `/tmp/deployment.` `flock` in Bash, or a storage system. If a d running, subsequent d wait or fail immediatel message.
The application version reported in `/health` does not match the newly deployed version after switch.	The deployment to the inactive environment succeeded, but the application process did not restart with the new code, or it's serving an old cached version (common in some Python/JavaScript frameworks).	1. Compare the version from the health endpoint of the newly active environment with the version you intended to deploy. 2. SSH into the server hosting the environment and check the deployed artifact (Docker image tag, or file timestamp). 3. Restart the application process manually and see if the version changes.	Ensure your `deploy` script **forces a clean** application. For contai this means stopping th starting a new one wit traditional deployment manager (systemd, su the service, and ensur code from disk (clear a module caches).

Debugging Techniques and Tools

Effective debugging requires a systematic approach and knowledge of the right tools to inspect each component's state. The following techniques form your primary toolkit.

1. Inspecting Environment State

The first step when something is wrong is to understand the current state of both blue and green environments independently.

- **Direct Health Checks:** Don't rely solely on the load balancer's health check. Use `curl` to query each environment's health endpoint directly.

```bash
# Check blue environment directly
curl -s http://localhost:8080/health | jq .
# Check green environment directly
curl -s http://localhost:8081/health | jq .
```

`Look for discrepancies in `version`, `database_connected`, and `status`. The `jq` tool nicely formats the JSON response.`

- **Process Inspection:** Verify the application processes are running and bound to the correct ports.

```bash
# List processes and their commanded line (look for your app)
ps aux | grep -E "(your-app-name|python|node|java)"
# Check which process is listening on the blue and green ports
sudo lsof -i :8080
sudo lsof -i :8081
sudo netstat -tlnp | grep -E ':(8080|8081)'
```

- **Log Aggregation:** Each environment should have its own dedicated log file. Tail these logs to see errors in real-time.

```bash
# Assuming logs are at /var/log/blue-app.log and /var/log/green-app.log
tail -f /var/log/blue-app.log
tail -f /var/log/green-app.log
```

2. Inspecting Load Balancer State

The load balancer is the traffic cop. If it's misconfigured, traffic goes to the wrong place or nowhere at all.

- **Verify Active Environment State:** Read the source of truth for which environment the load balancer *should* be routing to.

```bash
cat /tmp/active_environment
# Should output "blue" or "green"
```

- **Check Generated Configuration:** Examine the actual configuration file used by Nginx or HAProxy.

```bash
cat /etc/nginx/conf.d/bluegreen.conf
# Or for HAProxy
cat /etc/haproxy/haproxy.cfg
```

`Look for the `upstream` or `backend` definition. It should point to `localhost` and the port matching the environment in `ACTIVE_EN`

- **Test Configuration Syntax:** Before blaming the application, ensure the load balancer config is valid.

```bash
sudo nginx -t
# Output should be: "nginx: configuration file /etc/nginx/nginx.conf test is successful"
sudo haproxy -c -f /etc/haproxy/haproxy.cfg
```

- **Monitor Load Balancer Logs:** These logs show the traffic flow and any errors connecting to backends.

```bash
# Nginx error log
sudo tail -f /var/log/nginx/error.log
# Nginx access log (shows which backend was used for each request)
sudo tail -f /var/log/nginx/access.log
```

- **Trace a Request Through the Load Balancer:** Use `curl` with verbose output to see exactly where your request goes. You can also add a custom header to identify the request in the logs.

```bash
curl -v -H "X-Debug-ID: test123" http://your-loadbalancer-url/api/test
```

`Then, search for `test123` in both the load balancer access log and the application logs of blue and green to see which one handle`

3. Inspecting Deployment State

When an automated deployment fails, you need to reconstruct its steps.

- **Deployment Log File:** The orchestrator should log every significant action and its outcome to `DEPLOYMENT_LOG`.

```bash
tail -50 /var/log/deployments.log
```

`Look for error messages, exit codes, and the point where the process stopped.`

- **Check for Stale Artifacts or Containers:** A failed deployment might leave behind old containers or build artifacts that interfere with the next run.

```bash
# For Docker
docker ps -a | grep -E "(blue|green|your-app)"
docker images | grep your-app
# For file-based deployments
ls -la /opt/deployments/blue/ /opt/deployments/green/
```

4. Inspecting Database State

Database issues are often the trickiest to diagnose because they involve state that persists across deployments.

- **Verify Schema Version:** Connect to the database and check the `schema_migrations` table.

```bash
# Using the command-line client for your database, e.g., PostgreSQL
psql -U your_user -d your_db -c "SELECT * FROM schema_migrations ORDER BY applied_at DESC LIMIT 5;"
```

`Ensure the version matches what your application expects.`

- **Check for Pending Migrations:** Sometimes a migration script fails partially. Look for inconsistencies.

```bash
# Compare migration files on disk with applied versions
ls -1 migrations/ | sort
# Should match the list of versions in the database, perhaps with one extra (the pending one).
```

- **Test Backward Compatibility:** Manually verify that a simple query from the old application version would still work. For example, if you added a new column, ensure `SELECT` statements without that column still execute.
- **Monitor Database Connections:** During a switch, you might see a spike in connections from the new environment. Use database admin tools to see active connections and any long-running queries.

5. Systematic Troubleshooting Workflow

When faced with an undefined problem, follow this structured workflow:

1. **Identify the Symptom:** Precisely describe what the user is seeing (e.g., "502 error", "wrong data", "old version of the site").
2. **Locate the Faulty Component:** Use the **Request Journey** model. Start at the load balancer. Is it returning the error? If so, check its config and logs. If the load balancer is passing traffic, check the active environment's health and logs. If the app seems healthy, check the database.
3. **Check State Files:** Verify `ACTIVE_ENV_FILE`, deployment logs, and any other state tracking files. They are the "memory" of your automation and can reveal incorrect assumptions.
4. **Reproduce the Issue:** Can you reproduce it manually with `curl` or a browser? This helps isolate whether it's a problem with the automation or the underlying service.
5. **Review Recent Changes:** What was the last deployment? What migration ran? Check the commit history and deployment logs for the most recent change.
6. **Isolate the Environment:** Test the blue and green environments *directly* (bypassing the load balancer) to see if the issue is present in one, both, or neither.
7. **Rollback as a Diagnostic Tool:** If you suspect the new version, a controlled rollback using `rollback_deployment` can confirm the issue is version-specific. If the problem persists after rolling back, the issue is likely in the infrastructure or database.

> **Key Debugging Insight:** In a blue-green system, you have a powerful advantage—you almost always have a **known-good previous version** still running on the inactive side. Use this to your benefit. Compare the state, configuration, and behavior of the active (problematic) environment with the inactive (presumably working) environment. Differences between them are prime suspects for the root cause.

Implementation Guidance

This section provides concrete, executable tools and scripts to implement the debugging techniques described above. The primary language is Bash, as specified.

A. Technology Recommendations Table

Debugging Aspect	Simple Option (Bash/CLI)	Advanced Option
Health Checking	`curl` with `jq` for parsing JSON responses	Dedicated monitoring stack (Prometheus, Grafana) with custom exporters
Log Aggregation	`tail`, `grep`, `awk` on individual log files	Centralized logging (ELK Stack, Loki, Datadog) with structured JSON logs
Process Inspection	`ps`, `lsof`, `netstat`	Container orchestration dashboards (Kubernetes Dashboard, Docker Desktop)
Load Balancer Inspection	`nginx -t`, `haproxy -c`, manual config review	Load balancer APIs and dynamic configuration managers (Consul Template)
Database Inspection	Native CLI clients (`psql`, `mysql`), simple scripts	Database profiling tools (pgBadger, pt-query-digest), APM with DB tracing

B. Recommended File/Module Structure

Add a `scripts/debug/` directory to your project root to hold all debugging utilities. This keeps them organized and version-controlled.

```
project-root/
  scripts/
    deploy.sh                # Main deployment orchestrator
    loadbalancer/            # Load balancer controller scripts
    environments/            # Environment manager scripts
    migrations/              # Database migrator scripts
    debug/                   # <-- NEW: Debugging utilities
      check_health.sh        # Deep health check for an environment
      inspect_state.sh       # Prints current system state (active env, versions, etc.)
      trace_request.sh       # Helper to trace a request through the system
      view_logs.sh           # Tail relevant logs with color coding
```

C. Infrastructure Starter Code (Complete Debugging Scripts)

Here are complete, ready-to-use Bash scripts for common debugging tasks.

1. Deep Health Check Script (`scripts/debug/check_health.sh`): This script goes beyond a simple HTTP 200 check and validates critical application dependencies.

```bash
#!/usr/bin/env bash

# scripts/debug/check_health.sh

# Deep health check for a specific environment (blue or green)

# Usage: ./check_health.sh <blue|green>


set -euo pipefail


ENV_COLOR="${1:-}"

if [[ ! "$ENV_COLOR" =~ ^(blue|green)$ ]]; then

    echo "ERROR: Must specify 'blue' or 'green' as environment color."

    exit 1

fi


# Determine port based on environment

if [[ "$ENV_COLOR" == "blue" ]]; then

    PORT="8080"

else

    PORT="8081"

fi


HEALTH_URL="http://localhost:${PORT}/health"

echo "Performing deep health check for $ENV_COLOR environment on port $PORT..."

echo "============================================"


# 1. Check if the endpoint is reachable (with timeout)

if ! curl -f -s -m 10 "$HEALTH_URL" > /dev/null 2>&1; then

    echo "❌ FAIL: Health endpoint at $HEALTH_URL is unreachable or timed out."

    exit 2

fi


# 2. Get and parse the full health JSON

HEALTH_JSON=$(curl -s -m 10 "$HEALTH_URL")


# Use jq if available, otherwise use simple grep (fallback)

if command -v jq &> /dev/null; then

    echo "Health Response:"

    echo "$HEALTH_JSON" | jq .


    STATUS=$(echo "$HEALTH_JSON" | jq -r '.status')

    DB_CONNECTED=$(echo "$HEALTH_JSON" | jq -r '.database_connected')

    VERSION=$(echo "$HEALTH_JSON" | jq -r '.version')

    UPTIME=$(echo "$HEALTH_JSON" | jq -r '.uptime_seconds')

else

    echo "WARNING: 'jq' not installed. Parsing JSON crudely."
```

```
    STATUS=$(echo "$HEALTH_JSON" | grep -o '"status":"[^"]*"' | cut -d'"' -f4)
    DB_CONNECTED=$(echo "$HEALTH_JSON" | grep -o '"database_connected":[^,}]*' | cut -d':' -f2 | tr -d ' ')
    VERSION=$(echo "$HEALTH_JSON" | grep -o '"version":"[^"]*"' | cut -d'"' -f4)
    UPTIME=$(echo "$HEALTH_JSON" | grep -o '"uptime_seconds":[^,}]*' | cut -d':' -f2 | tr -d ' ')
fi

# 3. Evaluate health components
echo "----------------------------------------"
echo "Summary:"
echo "  Status:          $STATUS"
echo "  Database Connected: $DB_CONNECTED"
echo "  Version:         $VERSION"
echo "  Uptime (seconds): $UPTIME"

FAILED=0
if [[ "$STATUS" != "healthy" ]]; then
    echo "❌ FAIL: Overall status is not 'healthy'."
    FAILED=1
fi

if [[ "$DB_CONNECTED" != "true" ]]; then
    echo "❌ FAIL: Database connection is not healthy."
    FAILED=1
fi

if [[ -z "$VERSION" || "$VERSION" == "null" ]]; then
    echo "⚠️  WARN: Version information is missing."
fi

if [[ "$FAILED" -eq 0 ]]; then
    echo "✅ PASS: Deep health check passed for $ENV_COLOR."
    exit 0
else
    echo "❌ FAIL: Deep health check failed for $ENV_COLOR."
    exit 3
fi
```

2. System State Inspector (`scripts/debug/inspect_state.sh`): This script provides a single pane of glass for the current state of the entire blue-green system.

BASH

```bash
#!/usr/bin/env bash

# scripts/debug/inspect_state.sh

# Prints a comprehensive overview of the current blue-green deployment state.


set -euo pipefail


echo "========================================="

echo "Blue-Green Deployment System State Inspection"

echo "========================================="

echo "Timestamp: $(date -Iseconds)"

echo ""


# 1. Load Balancer Active Environment

echo "--- Load Balancer State ---"

if [[ -f "/tmp/active_environment" ]]; then

    ACTIVE_ENV=$(cat /tmp/active_environment 2>/dev/null || echo "unknown")

    echo "Active Environment (from $ACTIVE_ENV_FILE): $ACTIVE_ENV"

else

    echo "Active Environment file not found at $ACTIVE_ENV_FILE."

    ACTIVE_ENV="unknown"

fi


# 2. Check each environment's health

echo ""

echo "--- Environment Health ---"

for COLOR in blue green; do

    PORT=$([ "$COLOR" == "blue" ] && echo "8080" || echo "8081")

    HEALTH_URL="http://localhost:${PORT}/health"



    if curl -f -s -m 3 "$HEALTH_URL" > /dev/null 2>&1; then

        JSON=$(curl -s -m 3 "$HEALTH_URL")

        if command -v jq &> /dev/null; then

            VERSION=$(echo "$JSON" | jq -r '.version // "unknown"')

            STATUS=$(echo "$JSON" | jq -r '.status // "unknown"')

            DB=$(echo "$JSON" | jq -r '.database_connected // "unknown"')

        else

            VERSION=$(echo "$JSON" | grep -o '"version":"[^"]*"' | head -1 | cut -d'"' -f4 || echo "unknown")

            STATUS=$(echo "$JSON" | grep -o '"status":"[^"]*"' | head -1 | cut -d'"' -f4 || echo "unknown")

            DB=$(echo "$JSON" | grep -o '"database_connected":[^,}]*' | cut -d':' -f2 | tr -d ' ' || echo "unknown")

        fi

        MARKER=$([ "$COLOR" == "$ACTIVE_ENV" ] && echo " (ACTIVE)" || echo "")

        echo "✅ $COLOR$MARKER: Port $PORT, Version '$VERSION', Status '$STATUS', DB Connected: $DB"
```

```
    else
        MARKER=$([ "$COLOR" == "$ACTIVE_ENV" ] && echo " (ACTIVE)" || echo "")
        echo "❌ $COLOR$MARKER: Port $PORT - Health check failed or timed out."
    fi
done

# 3. Database Schema Version
echo ""
echo "--- Database Schema ---"
# This assumes you have a helper script or direct command to get the version.
# We'll simulate a check. In reality, integrate with your `get_current_schema_version` logic.
if command -v psql &> /dev/null; then
    # Example for PostgreSQL, adjust for your DB
    SCHEMA_VERSION=$(psql -U postgres -d app_db -t -c "SELECT version FROM schema_migrations ORDER BY applied_at DESC LIMIT 1;"
2>/dev/null || echo "Error querying")
    echo "Current Schema Version: $SCHEMA_VERSION"
else
    echo "Database client not available. Skipping schema version check."
fi

# 4. Recent Deployment Logs
echo ""
echo "--- Recent Deployment Activity (tail of $DEPLOYMENT_LOG) ---"
if [[ -f "/var/log/deployments.log" ]]; then
    tail -5 "/var/log/deployments.log" 2>/dev/null || echo "Could not read deployment log."
else
    echo "Deployment log file not found."
fi

echo ""
echo "========================================="
```

D. Core Logic Skeleton Code (For Integration into Components)

These are the key functions you should ensure exist in your main components to aid debuggability. The TODOs guide you to add proper logging and state exposure.

1. Enhanced `check_health` in Environment Manager (`scripts/environments/health.sh`):

```bash
#!/usr/bin/env bash

# scripts/environments/health.sh

# check_health(env_color) returns boolean

# Performs a deep health assessment of the specified environment.


check_health() {

    local env_color="$1"

    local port

    # TODO 1: Determine the correct port for the environment (BLUE_PORT or GREEN_PORT)

    # TODO 2: Perform a basic HTTP GET to /health endpoint, timeout after 5 seconds

    # TODO 3: If HTTP fails (non-200 or timeout), log error and return false

    # TODO 4: Parse the JSON response. Extract `status`, `database_connected`, and `version`

    # TODO 5: Validate that status is "healthy" AND database_connected is true

    # TODO 6: Log detailed health metrics for debugging (uptime, timestamp)

    # TODO 7: Return true only if all deep checks pass. Return false otherwise.

}
```

2. Enhanced `switch_traffic` with Post-Switch Validation (`scripts/loadbalancer/switch.sh`):

```bash
#!/usr/bin/env bash

# scripts/loadbalancer/switch.sh

# switch_traffic(target_env) returns void

# Updates load balancer to route to target environment after health validation.


switch_traffic() {

    local target_env="$1"

    # TODO 1: Validate target_env is "blue" or "green"

    # TODO 2: Call check_health(target_env) and abort if unhealthy

    # TODO 3: Generate new load balancer config for target_env using generate_config_for_env

    # TODO 4: Validate the new config using validate_load_balancer_config (e.g., nginx -t)

    # TODO 5: Write the new active environment to ACTIVE_ENV_FILE

    # TODO 6: Perform a graceful reload of the load balancer (e.g., nginx -s reload)

    # TODO 7: Verify the reload succeeded by checking the process return code

    # TODO 8: (CRITICAL FOR DEBUGGING) Perform a post-switch smoke test by making a request to the load balancer's public IP and verifying it reaches the target environment. Log the result.

    # TODO 9: If any step fails, log the error, revert ACTIVE_ENV_FILE if changed, and exit with error.

}
```

E. Language-Specific Hints (Bash)

- **Error Handling:** Use `set -euo pipefail` at the top of your scripts. This makes the script exit on any error, undefined variable, or pipe failure.
- **Logging:** Don't just `echo`. Use a function that writes to both stdout and a log file with timestamps.

```bash
log() {
    local message="$1"
    echo "$(date -Iseconds) - $message" | tee -a "$DEPLOYMENT_LOG"
}
```

- **JSON Parsing:** Install `jq`. It's indispensable for parsing health check responses and configuration. If you can't install it, use `grep` and `cut` as a fragile fallback.
- **Timeouts:** Always use `timeout` or `curl --max-time` for any network call in a script to prevent hanging.
- **Checking Exit Codes:** Always check `$?` after critical commands (like `docker run`, `nginx -s reload`) and log success/failure.

F. Debugging Tips Table (Structured as in Implementation Guidance)

Symptom	Likely Cause	How to Diagnose	Fix
`check_health.sh` passes but users see 500 errors.	Health endpoint checks are superficial (only process running).	Run the app's functional API endpoints directly against the environment port. Check application logs for runtime exceptions.	Implement **deep health checks** that call a critical internal API or verify a database query succeeds.
`inspect_state.sh` shows "Active Environment: green" but traffic is still going to blue.	Load balancer config not reloaded, or reload failed silently.	Run `sudo nginx -T` to see the running config. Check `error.log` for reload errors.	In `switch_traffic`, add explicit validation: after reload, query the load balancer admin socket or verify the new config is active.
Database migration fails with "duplicate key" in `schema_migrations`.	Migration scripts are not idempotent; the same version was applied twice.	Check the `schema_migrations` table for duplicate version entries.	Make migration scripts idempotent using `CREATE TABLE IF NOT EXISTS` or `ALTER TABLE ... ADD COLUMN IF NOT EXISTS`. Use transactions where possible.
Deployment log shows "Smoke test passed" but users report bugs.	Smoke tests are not comprehensive enough; they hit a simple endpoint but don't test critical user journeys.	Review your smoke test suite. Does it test the main user flows?	Expand smoke tests to cover critical business workflows (login, purchase, data submission). Consider using a headless browser or more advanced API testing.

G. Milestone Checkpoint (Debugging)

After implementing the debugging scripts and enhanced logging, verify your debugging capabilities with the following checkpoint:

1. **Run the system state inspector:**

```bash
bash scripts/debug/inspect_state.sh
```

`**Expected Output:** A clear table showing the active environment, health status of both blue and green, their versions, database c`

2. **Simulate a failure and diagnose it:**
   - Stop the database temporarily.
   - Run `bash scripts/debug/check_health.sh blue` **Expected Output:** The script should output a clear failure message indicating the database connection is not healthy, and exit with a non-zero code.
3. **Test request tracing:**
   - Add a unique header to a request: `curl -H "X-Request-ID: debug-test-001" http://localhost`
   - Immediately check the load balancer access log and the active environment's application log for that ID. **Expected Outcome:** You should find the `X-Request-ID` in both logs, confirming you can trace a request through the system.
4. **Verify deployment logging:**
   - Run a deployment (or a dry-run).
   - Check the deployment log: `tail -f /var/log/deployments.log` **Expected Outcome:** Each significant step (build start, deploy to environment, health check, smoke test, switch) is logged with a timestamp and success/failure indication.

Signs something is wrong:

- The state inspector cannot read the active environment file → Check permissions and path.
- Health checks pass even when the database is down → Your health endpoint is not checking the database deeply.
- Logs don't contain the `X-Request-ID` → Your application or load balancer is not configured to pass through or log custom headers.

Future Extensions

> **Milestone(s):** Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

The blue-green deployment architecture we've built provides a solid foundation for zero-downtime releases, but it's designed to be extensible. Think of it as a **modular construction set**—the basic components (environments, load balancer, orchestrator, migrator) form a stable platform upon which we can build increasingly sophisticated deployment capabilities. This section explores potential enhancements that the architecture can accommodate without requiring fundamental redesign.

Possible Enhancements

Automated Canary Analysis

Mental Model: Test Driving a Car Before Buying Instead of immediately switching all traffic to the new version, imagine taking the new environment for a "test drive" with a small percentage of real users. You monitor key metrics (error rates, latency, business KPIs) and only proceed with full rollout if the new version performs at least as well as the old one. This is canary analysis—a gradual, data-driven validation of new releases.

Description: Canary analysis extends our binary "all-or-nothing" traffic switch with a gradual rollout capability. The system would:

1. Initially route a small percentage (e.g., 1-5%) of traffic to the green environment after deployment
2. Monitor predefined metrics (error rates, response times, business metrics) from both environments
3. Automatically increase traffic percentage to green if metrics remain within acceptable bounds
4. Automatically roll back if metrics degrade beyond thresholds

Implementation Considerations:

- **Metrics Collection:** Requires integration with monitoring systems (Prometheus, Datadog, CloudWatch)
- **Traffic Splitting:** The load balancer must support weighted routing (Nginx weight parameter, HAProxy weight)
- **Decision Engine:** Needs rules for evaluating metrics and making rollout/rollback decisions
- **State Management:** Must track current canary stage (percentage, duration, evaluation results)

Architecture Decision: Canary Analysis Strategy

> **Decision: Gradual Traffic Ramp with Automated Rollback**
>
> - **Context:** Basic blue-green deployments switch all traffic at once, which exposes all users to potential issues simultaneously. We need a safer, more gradual validation mechanism.
> - **Options Considered:**
>   1. **Time-based canary:** Increase traffic percentage on a fixed schedule regardless of metrics
>   2. **Metric-based canary:** Increase traffic only when metrics meet acceptance criteria
>   3. **Manual canary:** Require human approval at each traffic percentage increment
> - **Decision:** Implement metric-based canary analysis with automated rollback
> - **Rationale:** Time-based canaries risk exposing users to broken versions. Manual canaries add operational overhead. Metric-based automation provides safety while maintaining deployment velocity.
> - **Consequences:** Requires robust monitoring infrastructure, adds complexity to deployment orchestration, but significantly reduces risk of production incidents.

Option	Pros	Cons	Suitability
Time-based canary	Simple to implement, predictable timeline	Ignores actual performance, high risk if issues appear late	Low - too risky for production
Metric-based canary	Data-driven, automatically aborts on degradation	Requires comprehensive monitoring, more complex	High - balances safety and automation
Manual canary	Human oversight at each stage, flexible	Slow, adds operational burden, prone to human error	Medium - good for initial implementation

Integration with Feature Flags

Mental Model: Light Switches with Dimmer Controls Feature flags are like light switches for code paths, but with dimmer controls. Instead of deploying all-or-nothing, you can deploy code with certain features "switched off" at the infrastructure level, then toggle them on for specific user segments, time periods, or percentages of traffic—all without another deployment.

Description: Feature flag integration allows decoupling deployment from feature activation. The system would:

1. Deploy new code containing feature flags to both blue and green environments

2. Use a feature flag service (LaunchDarkly, Flagsmith, or custom) to control feature activation
3. Enable features progressively (by user segment, percentage, geography) while monitoring impact
4. Roll back features instantly without redeploying code by toggling flags

Implementation Considerations:

- **Flag Evaluation:** Application code must check feature flag status at runtime
- **Flag Management:** Requires feature flag service integration or built-in flag management
- **Consistency:** Feature flag states must be consistent across blue and green environments during traffic switches
- **Cleanup:** Need processes for removing old feature flags from code

Integration Points with Current Architecture:

- **Deployment Orchestrator:** Can validate that feature flag configurations are synchronized before traffic switch
- **Environment Manager:** Should inject feature flag service credentials/endpoints via environment variables
- **Database Migrator:** Feature flags can control database migration application (e.g., run new schema code only when flag is enabled)

Multi-Region Deployment

Mental Model: Air Traffic Control for Multiple Airports Our current architecture manages two runways at one airport. Multi-region deployment extends this to manage synchronized deployments across multiple geographically distributed "airports," with traffic routing that considers user location, region health, and data sovereignty requirements.

Description: Multi-region deployment extends blue-green across geographic boundaries for disaster recovery, reduced latency, and regulatory compliance. The system would:

1. Maintain blue-green pairs in multiple regions (e.g., us-east-1, eu-west-1, ap-southeast-1)
2. Coordinate deployments across regions (sequential, parallel, or wave-based)
3. Route users to the nearest healthy region with active environment
4. Handle database replication and migration coordination across regions

Implementation Considerations:

- **Global Traffic Management:** Requires DNS-based (Route53, Cloudflare) or Anycast routing
- **Data Consistency:** Needs cross-region database replication strategy (active-active, active-passive)
- **Deployment Coordination:** Orchestrator must manage deployment sequencing across regions
- **Failover Automation:** Automatic regional failover when a region becomes unhealthy

Regional Deployment Strategies:

Strategy	Description	Pros	Cons
Sequential	Deploy to one region at a time, validate, then proceed to next	Lower risk, easier to manage, isolate issues	Slower global rollout, longer time to value
Parallel	Deploy to all regions simultaneously	Fastest global rollout	Risk multiplies, harder to coordinate rollback
Wave-based	Deploy to regions in groups (wave 1: primary regions, wave 2: secondary)	Balances speed and risk, allows validation between waves	More complex orchestration

Auto-Rollback Based on Metrics

Mental Model: Car's Automatic Emergency Braking Current rollback requires manual trigger or smoke test failure detection. Auto-rollback is like a car's automatic emergency braking—continuously monitoring key metrics and instantly reverting when thresholds are breached, without waiting for human intervention.

Description: Auto-rollback extends the basic rollback capability with proactive failure detection. The system would:

1. Continuously monitor key performance indicators (KPIs) after traffic switch
2. Compare metrics between pre-switch baseline and post-switch performance
3. Automatically trigger rollback if metrics breach predefined thresholds (error rate > 1%, latency increase > 50%, etc.)
4. Implement progressive degradation handling (immediate rollback for critical failures, alert-only for minor issues)

Key Metrics to Monitor:

- **Error rates:** HTTP 5xx errors, application exceptions
- **Latency:** P50, P95, P99 response times
- **Throughput:** Requests per second, successful transactions
- **Business metrics:** Conversion rates, cart abandonment, revenue

- **Infrastructure metrics:** CPU, memory, database connections

Implementation Architecture:

Component	Enhanced Responsibility	Integration Point
Deployment Orchestrator	Post-switch monitoring phase, auto-rollback trigger	Extends `rollback_deployment()` with metric validation
Environment Manager	Enhanced health checks with business metrics	`check_health()` returns structured metrics, not just boolean
Load Balancer Controller	Traffic splitting for A/B metric comparison	`generate_config_for_env()` supports percentage-based routing
External Systems	Metric collection and analysis (Prometheus, Datadog)	Webhook callbacks or API polling

Database Migration with Online Schema Change Tools

Mental Model: Underwater Welding for Live Structures Our current expand-contract pattern works well for simple migrations but becomes complex for large tables or multiple changes. Online schema change tools are like underwater welding—making structural changes to the database while it remains operational and serving traffic.

Description: Integrate specialized tools (pt-online-schema-change, gh-ost, Liquidbase, Flyway) that automate complex schema migrations without locking tables or requiring extended maintenance windows. The system would:

1. Use these tools for migrations that are complex or risky to implement manually
2. Integrate migration tool execution into the deployment pipeline
3. Monitor migration progress and automatically pause/rollback if issues arise
4. Coordinate application deployment timing with migration completion

Tool Comparison:

Tool	Mechanism	Pros	Cons	Best For
pt-online-schema-change	Creates shadow table, copies data, swaps	No locking, battle-tested	Requires triggers, some performance impact	MySQL, single complex migrations
gh-ost	Binlog-based, no triggers	No trigger overhead, pause/resume	MySQL only, more complex setup	Large MySQL tables
Liquibase/Flyway	Versioned SQL scripts	Declarative, tracks history, multi-database	May require downtime for some operations	Cross-database consistency, team workflows

Blue-Green for Stateful Services

Mental Model: Hot-Swappable Battery Packs Our current architecture assumes stateless application services. Stateful services (caches, WebSocket connections, file uploads) are like devices with internal batteries—they can't be simply switched without losing state. Blue-green for stateful services requires "hot-swappable" state transfer.

Description: Extend blue-green deployment to services that maintain client state or local data. The system would:

1. Implement state synchronization or replication between blue and green environments
2. Use connection draining to gracefully migrate stateful connections
3. Coordinate state transfer during traffic switch (session replication, cache warming)
4. Handle stateful protocols (WebSockets, long-polling, SSE) with graceful migration

State Transfer Strategies:

State Type	Transfer Strategy	Implementation Complexity	Downtime Risk
User sessions	Session replication or external session store (Redis)	Low-Medium	Low
WebSocket connections	Graceful closure with reconnection guidance	Medium-High	Medium (brief disruption)
Local file uploads	Shared storage or replication to both environments	Medium	None with shared storage
In-memory cache	Cache warming or shared cache layer	Low-Medium	Performance degradation during warmup

Cost Optimization: Dynamic Environment Scaling

Mental Model: Hotel Room Management Currently, both blue and green environments run continuously, like maintaining two fully staffed hotel wings even when only one is occupied. Cost optimization dynamically scales the inactive environment based on usage patterns.

Description: Reduce infrastructure costs by scaling down (or stopping) the inactive environment when not in use, and scaling it up before deployment. The system would:

1. Monitor traffic patterns and deployment schedules
2. Scale down inactive environment during low-traffic periods
3. Automatically scale up inactive environment before scheduled deployments
4. Implement warm-up procedures (cache warming, connection pooling) before traffic switch

Scaling Strategies:

Strategy	Implementation	Cost Savings	Trade-off
Stop/Start	Completely stop inactive environment VM/container	Maximum savings	Longer deployment time (cold start)
Scale Down	Reduce replica count to minimum (1)	Significant savings	Some cost remains, faster warm-up
Pause/Resume	Container pause/unpause or VM suspend/resume	Good savings	State preserved, moderate restart time

Implementation Guidance

Technology Recommendations

Enhancement	Simple Option	Advanced Option
Automated Canary Analysis	Script-based metric polling with simple thresholds	Integration with Prometheus + Grafana for real-time analysis
Feature Flags	Environment variable-based flags	External service (LaunchDarkly, Flagsmith) with SDK integration
Multi-Region Deployment	Manual region-by-region deployment scripts	Terraform/CloudFormation templates with deployment coordination
Auto-Rollback	Post-deploy script with basic metric checks	Integration with APM (New Relic, Datadog) webhooks
Database Migration Tools	Manual pt-online-schema-change execution	Integrated migration runner with progress monitoring
Stateful Services	Shared external state (Redis, S3)	State synchronization service with conflict resolution
Cost Optimization	Scheduled scaling scripts	Autoscaling based on deployment calendar and metrics

Recommended File Structure for Extensions

```
blue-green-system/
├── extensions/                  # Future enhancement modules
│   ├── canary/
│   │   ├── canary_analyzer.sh    # Metric analysis and decision logic
│   │   ├── metrics_collector.py  # Fetches metrics from monitoring
│   │   └── weighted_routing.conf # Nginx config for traffic splitting
│   ├── feature_flags/
│   │   ├── flag_evaluator.py     # Evaluates feature flag states
│   │   ├── flag_sync.sh          # Synchronizes flags across environments
│   │   └── flags.json            # Feature flag definitions
│   ├── multi_region/
│   │   ├── region_coordinator.sh # Coordinates deployments across regions
│   │   ├── dns_updater.py        # Updates global DNS for failover
│   │   └── regions.yaml          # Region configuration
│   └── auto_rollback/
│       ├── metric_monitor.py     # Continuously monitors KPIs
│       ├── rollback_trigger.sh   # Triggers rollback based on metrics
│       └── thresholds.yaml       # Metric thresholds for auto-rollback
├── scripts/                     # Core scripts (existing)
│   ├── deploy.sh
│   ├── switch_traffic.sh
│   └── rollback.sh
└── config/                      # Configuration (existing)
    ├── nginx.conf
    └── environments.yaml
```

Canary Analysis Starter Implementation

Simple Canary Analyzer (Bash):

BASH

```bash
#!/bin/bash

# canary_analyzer.sh

# Simple canary analysis with metric thresholds


CANARY_DURATION=${1:-300}  # 5 minutes default

CANARY_STEPS=${2:-5}       # 5 traffic increase steps

BASELINE_ERROR_RATE=${3:-0.01}  # 1% baseline error rate

MAX_ERROR_RATE=${4:-0.05}  # 5% maximum allowed during canary


# Function to get current error rate from metrics endpoint

get_error_rate() {

    local env_color=$1

    local port=$([ "$env_color" = "blue" ] && echo $BLUE_PORT || echo $GREEN_PORT)



    # Query metrics endpoint (simplified - in reality would use Prometheus API)

    curl -s "http://localhost:$port/metrics" | \

        grep 'http_requests_total{status=~"5.."}' | \

        awk '{print $2}'  # Returns error rate

}


# Function to increase traffic percentage

increase_traffic_percentage() {

    local percentage=$1

    echo "Increasing traffic to green to $percentage%"



    # Generate new Nginx config with weighted routing

    cat > /tmp/nginx_canary.conf << EOF

upstream backend {

    server localhost:$BLUE_PORT weight=${$((100 - percentage))};

    server localhost:$GREEN_PORT weight=$percentage;

}


server {

    listen 80;

    location / {

        proxy_pass http://backend;

    }

}

EOF



    # Apply config

    sudo cp /tmp/nginx_canary.conf /etc/nginx/conf.d/default.conf
```

```
    sudo nginx -s reload
}

# Main canary analysis loop
perform_canary_analysis() {
    local step_duration=$((CANARY_DURATION / CANARY_STEPS))
    local current_percentage=0

    for step in $(seq 1 $CANARY_STEPS); do
        current_percentage=$((step * 20))  # 20%, 40%, 60%, 80%, 100%

        echo "Starting canary step $step: $current_percentage% traffic for $step_duration seconds"
        increase_traffic_percentage $current_percentage

        # Monitor for the step duration
        local start_time=$(date +%s)
        while [ $(($(date +%s) - start_time)) -lt $step_duration ]; do
            local error_rate=$(get_error_rate "green")

            # Check if error rate exceeds threshold
            if (( $(echo "$error_rate > $MAX_ERROR_RATE" | bc -l) )); then
                echo "ERROR: Error rate $error_rate exceeds maximum $MAX_ERROR_RATE"
                echo "Triggering rollback..."
                rollback_deployment "green"
                exit 1
            fi

            sleep 10  # Check every 10 seconds
        done

        echo "Canary step $step completed successfully"
    done

    echo "Canary analysis completed successfully. Green environment is healthy."
    return 0
}
```

Feature Flag Integration Skeleton

Feature Flag Manager (Python skeleton):

PYTHON

```python
#!/usr/bin/env python3

# feature_flag_manager.py

"""

Feature flag manager for blue-green deployments.

Integrates with external flag services or manages local flags.

"""


import json

import os

from typing import Dict, Any


class FeatureFlagManager:

    def __init__(self, config_path: str = "config/feature_flags.json"):

        self.config_path = config_path

        self.flags = self.load_flags()


    def load_flags(self) -> Dict[str, Any]:

        """TODO 1: Load feature flags from configuration file or API"""

        # TODO: Implement flag loading from JSON file

        # TODO: Support environment-specific flag variations

        # TODO: Cache flags with TTL for performance

        return {}


    def is_enabled(self, flag_name: str, user_context: Dict = None) -> bool:

        """TODO 2: Check if a feature flag is enabled for given context"""

        # TODO: Get flag definition from self.flags

        # TODO: Evaluate flag rules based on user_context

        # TODO: Handle percentage rollouts, user segments, etc.

        # TODO: Return True/False based on evaluation

        return False


    def sync_flags_across_environments(self, blue_endpoint: str, green_endpoint: str) -> bool:

        """TODO 3: Ensure feature flags are consistent across blue and green"""

        # TODO: Fetch flags from blue environment

        # TODO: Fetch flags from green environment

        # TODO: Compare and identify differences

        # TODO: Synchronize if necessary

        # TODO: Return True if synchronized, False if conflicts found

        return True


    def validate_flag_readiness(self, environment: str) -> bool:
```

```
        """TODO 4: Validate that all required feature flags are available"""
        # TODO: Check connectivity to flag service
        # TODO: Verify critical flags are defined
        # TODO: Test flag evaluation for sample contexts
        # TODO: Return True if ready for traffic
        return True

# TODO 5: Integrate with deployment orchestrator
# - Call sync_flags_across_environments() before traffic switch
# - Call validate_flag_readiness() during smoke tests
# - Add feature flag validation to deployment pre-conditions
```

Multi-Region Coordinator Skeleton

Region Deployment Coordinator (Bash skeleton):

```bash
#!/bin/bash

# region_coordinator.sh

# Coordinates blue-green deployments across multiple regions


REGIONS=("us-east-1" "eu-west-1" "ap-southeast-1")

PRIMARY_REGION="us-east-1"

DEPLOYMENT_WAVES=(

    "us-east-1"

    "eu-west-1 ap-southeast-1"

)


deploy_to_region() {

    local region=$1

    local env_color=$2



    echo "Deploying to region: $region, environment: $env_color"



    # TODO 1: Set region-specific environment variables

    export AWS_REGION=$region

    export DB_HOST="db-$region.example.com"



    # TODO 2: Trigger deployment in target region

    # This could be via SSH, AWS SSM, or region-specific CI/CD pipeline

    # ssh deploy@$region "./deploy.sh --environment $env_color"



    # TODO 3: Wait for deployment completion

    # while ! check_region_health $region $env_color; do sleep 5; done



    echo "Deployment to $region completed"

}


check_region_health() {

    local region=$1

    local env_color=$2



    # TODO 4: Check health endpoint in specific region

    # curl -f "https://$env_color.$region.example.com/health"



    return 0  # Placeholder

}


switch_region_traffic() {
```

```
    local region=$1
    local target_env=$2

    echo "Switching traffic in region $region to $target_env"

    # TODO 5: Update region-specific load balancer or DNS
    # aws route53 change-resource-record-sets ...
    # or update region's load balancer target group

    # TODO 6: Verify traffic switch completion
    # Wait for DNS propagation or load balancer update
}

deploy_wave() {
    local wave_num=$1
    shift
    local regions=("$@")

    echo "Starting deployment wave $wave_num to regions: ${regions[*]}"

    # Deploy to all regions in this wave in parallel
    local pids=()
    for region in "${regions[@]}"; do
        deploy_to_region "$region" "green" &
        pids+=($!)
    done

    # Wait for all deployments in wave to complete
    for pid in "${pids[@]}"; do
        wait $pid
    done

    # Validate all regions in wave
    for region in "${regions[@]}"; do
        if ! check_region_health "$region" "green"; then
            echo "ERROR: Region $region failed health check"
            return 1
        fi
    done

    echo "Wave $wave_num deployment successful"
```

```
}

# Main multi-region deployment flow
perform_multi_region_deployment() {
    echo "Starting multi-region blue-green deployment"

    # Wave-based deployment
    local wave_num=1
    for wave_regions in "${DEPLOYMENT_WAVES[@]}"; do
        if ! deploy_wave $wave_num $wave_regions; then
            echo "ERROR: Wave $wave_num failed. Initiating rollback..."
            # TODO: Implement cross-region rollback
            return 1
        fi

        # Switch traffic for successful wave
        for region in $wave_regions; do
            switch_region_traffic "$region" "green"
        done

        ((wave_num++))
    done

    echo "Multi-region deployment completed successfully"
    return 0
}
```

Auto-Rollback Trigger Implementation

Metric-Based Rollback Trigger (Bash/Python hybrid):

```bash
#!/bin/bash

# auto_rollback_trigger.sh

# Monitors metrics and triggers rollback if thresholds breached


MONITOR_DURATION=${1:-600}  # Monitor for 10 minutes post-deployment

CHECK_INTERVAL=${2:-30}     # Check metrics every 30 seconds


# Metric thresholds (would normally come from config)

THRESHOLDS=(

    "error_rate:0.05"      # 5% maximum error rate

    "p95_latency:2000"     # 2 seconds maximum P95 latency

    "throughput:0.5"       # Minimum 50% of baseline throughput

)


# Function to fetch metric value

fetch_metric() {

    local metric_name=$1

    local env_color=$2



    # TODO 1: Implement actual metric fetching

    # Example: Query Prometheus API

    # curl -s "http://prometheus:9090/api/v1/query?query=$metric_name{env='$env_color'}[1m]" | jq '.data.result[0].value[1]'



    # Placeholder: return random value for demonstration

    case $metric_name in

        error_rate) echo "0.03" ;;

        p95_latency) echo "1500" ;;

        throughput) echo "0.8" ;;

        *) echo "0" ;;

    esac

}


# Function to check if metric breaches threshold

check_metric_threshold() {

    local metric_name=$1

    local current_value=$2

    local threshold=$3



    # Different comparison logic for different metrics

    case $metric_name in

        error_rate|p95_latency)

            # Higher is worse
```

```
            if (( $(echo "$current_value > $threshold" | bc -l) )); then
                return 1  # Breached
            fi
            ;;
        throughput)
            # Lower is worse
            if (( $(echo "$current_value < $threshold" | bc -l) )); then
                return 1  # Breached
            fi
            ;;
    esac

    return 0  # Within threshold
}

# Main monitoring loop
monitor_and_rollback() {
    local active_env=$1
    local start_time=$(date +%s)

    echo "Starting post-deployment monitoring for $active_env environment"
    echo "Will monitor for $MONITOR_DURATION seconds, checking every $CHECK_INTERVAL seconds"

    while [ $(($(date +%s) - start_time)) -lt $MONITOR_DURATION ]; do
        echo "Checking metrics at $(date)"

        local all_ok=true

        # Check each threshold
        for threshold_spec in "${THRESHOLDS[@]}"; do
            IFS=':' read -r metric_name threshold <<< "$threshold_spec"
            current_value=$(fetch_metric "$metric_name" "$active_env")

            if ! check_metric_threshold "$metric_name" "$current_value" "$threshold"; then
                echo "ALERT: $metric_name breached! Value: $current_value, Threshold: $threshold"
                all_ok=false

                # For critical metrics, trigger immediate rollback
                if [[ "$metric_name" == "error_rate" ]] && (( $(echo "$current_value > 0.1" | bc -l) )); then
                    echo "CRITICAL: Error rate > 10%. Triggering emergency rollback!"
                    rollback_deployment "$active_env"
```

```
                    return 1
                fi
            else
                echo "OK: $metric_name = $current_value (threshold: $threshold)"
            fi
        done

        if [ "$all_ok" = false ]; then
            echo "WARNING: Some metrics breached thresholds. Monitoring will continue..."
            # TODO: Implement progressive degradation handling
            # Could wait for consecutive failures before rollback
        fi

        # Wait for next check
        sleep $CHECK_INTERVAL
    done

    echo "Monitoring period completed. $active_env environment is stable."
    return 0
}

# TODO 2: Integrate with deployment orchestrator
# Call monitor_and_rollback() after successful traffic switch
# Run in background to avoid blocking deployment pipeline
# monitor_and_rollback "green" &
```

Language-Specific Hints for Extensions

Bash-Specific Tips:

- Use `bc` for floating-point arithmetic in threshold comparisons
- Implement timeouts with `timeout` command or `$SECONDS` variable
- Use `trap` to catch signals and implement graceful shutdown of monitoring processes
- Consider using `jq` for JSON parsing when integrating with API endpoints

Python Integration Points:

- Use `requests` library for API calls to monitoring systems
- Implement retry logic with exponential backoff for external service calls
- Use `threading` or `asyncio` for parallel region deployments
- Consider `prometheus-client` for exposing application metrics

Go Integration (if extending beyond Bash):

- Use `context` with timeouts for all external calls
- Implement circuit breakers for feature flag service calls
- Use `sync.WaitGroup` for coordinating parallel operations
- Consider `viper` for configuration management across regions

Milestone Checkpoint for Extensions

After implementing canary analysis:

```bash
# Run canary deployment
./extensions/canary/canary_deploy.sh --version 2.0.0 --traffic-steps 20,40,60,80,100

# Expected output:
# Starting canary deployment of version 2.0.0
# Deploying to green environment...
# Starting canary step 1: 20% traffic for 60 seconds
# Canary step 1 completed successfully
# Starting canary step 2: 40% traffic for 60 seconds
# ...
# Canary analysis completed successfully. Green environment is healthy.
# Switching 100% traffic to green...
```

After implementing feature flags:

```bash
# Test feature flag synchronization
./extensions/feature_flags/flag_sync.sh --environment blue --environment green

# Expected output:
# Checking feature flag consistency...
# Flags synchronized: 100%
# All critical flags available in both environments
# Feature flags ready for deployment
```

Verification steps:

1. Deploy with canary analysis enabled, simulate high error rate, verify auto-rollback
2. Toggle feature flags during deployment, verify behavior changes without redeployment
3. Simulate region failure during multi-region deployment, verify failover

Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
Canary traffic not splitting correctly	Nginx weight syntax error or missing reload	Check nginx error logs: `sudo nginx -t` and `sudo tail -f /var/log/nginx/error.log`	Verify weight syntax, ensure nginx reloaded not restarted
Feature flags inconsistent across environments	Flag service replication delay or caching	Compare flag states: `curl blue:8080/flags` vs `curl green:8081/flags`	Implement flag synchronization step before traffic switch
Multi-region deployment hangs on one region	Region-specific infrastructure issue or timeout	Check region deployment logs, verify network connectivity to region	Implement region deployment timeouts, add skip-failed-region option
Auto-rollback triggers too aggressively	Metric thresholds too sensitive or noisy data	Examine metric graphs around rollback time, check for data spikes	Adjust thresholds, add requirement for consecutive breaches
Cost optimization causes slow deployments	Cold start time for scaled-down environment	Measure environment startup time vs traffic warm-up time	Implement pre-warm before deployment, adjust scale-down aggressiveness

These extensions demonstrate how the modular blue-green architecture can evolve to handle increasingly sophisticated deployment scenarios while maintaining the core benefits of zero-downtime and instant rollback.

Glossary

> **Milestone(s):** Milestone 1 (Dual Environment Setup), Milestone 2 (Load Balancer Switching), Milestone 3 (Deployment Automation), Milestone 4 (Rollback & Database Migrations)

This glossary provides clear definitions for key terms, acronyms, and domain-specific vocabulary used throughout this design document. Think of it as a reference dictionary that helps establish a shared vocabulary for your team when discussing blue-green deployment concepts—much like a flight crew uses standardized terminology to ensure clear communication during critical operations.

Clear terminology prevents misunderstandings that can lead to deployment failures. For example, confusing "restart" with "reload" might cause dropped connections during a traffic switch. Similarly, misunderstanding "backward compatibility" could lead to database schema changes that break the previous application version during rollback.

Terminology

Term	Definition	Related Concepts
Active Environment	The production environment currently receiving 100% of live user traffic through the load balancer. This is the "live" environment that users interact with.	`get_current_active_env()`, Traffic Switch, Load Balancer
Artifact	An immutable, versioned package containing the application code, dependencies, and configuration ready for deployment. Examples: Docker image, JAR file, or compiled binary.	Versioning, Immutable Infrastructure, `deploy_to_environment()`
Atomic Switch	A traffic redirection operation that either fully succeeds (all traffic goes to the target environment) or fully fails (traffic stays on the current environment) with no intermediate states visible to users.	`switch_traffic()`, Idempotent, Transactionality
Backward Compatibility	The property where older (previous) application versions continue to function correctly with a newer database schema. This enables instant rollback by ensuring the previous environment's code still works after schema changes.	Expand-Contract Pattern, Database Migration, `validate_migration_compatibility()`
Blue Environment	One of the two identical production environments, typically configured to run on port `BLUE_PORT` (8080). The color "blue" is an arbitrary label—it could be called "A" and "B" or "production-1" and "production-2."	`BLUE_PORT`, Environment Identity, Dual Environment
Blue-Green Deployment	A deployment strategy that maintains two identical production environments (blue and green). Only one environment serves live traffic at a time, allowing safe deployment and testing in the inactive environment before switching traffic.	Zero-Downtime, Traffic Switch, Dual Environment Setup
Canary Analysis	A gradual, data-driven validation approach where a new release is exposed to a small percentage of users while monitoring key metrics (error rates, latency). If metrics degrade, the release is rolled back before affecting all users.	Gradual Rollout, Feature Flags, Auto-Rollback
Connection Draining	The process of allowing existing, in-flight connections to complete their work before terminating a process or removing a server from service. This prevents abrupt connection termination during deployments or traffic switches.	Graceful Shutdown, Graceful Reload, Zero-Downtime
Contract Phase	The final stage of the expand-contract migration pattern where old database columns, tables, or constraints that are no longer needed by the new application version are safely removed. This phase should only run after the new code is fully deployed and verified.	Expand-Contract Pattern, Database Migration, Backward Compatibility
Database Migration	A scripted, versioned change to the database schema (adding tables, columns, indexes) or reference data. Migrations are applied idempotently and tracked in a version table like `schema_migrations`.	`apply_migrations()`, `MIGRATIONS_DIR`, `VERSION_TABLE`
Deployment Orchestrator	The central automation component that coordinates the entire deployment sequence: building artifacts, deploying to the inactive environment, running smoke tests, and triggering the traffic switch.	`deploy_to_environment()`, `run_smoke_tests()`, `switch_traffic()`
Dual Environment Setup	The infrastructure foundation of blue-green deployment: two identical, isolated environments (blue and green) that can run simultaneously on separate ports or hostnames.	Milestone 1, Environment Isolation, `start_environment()`
Environment Identity	The unique identifier that distinguishes blue from green environments, typically implemented via environment variables (`ENVIRONMENT_COLOR`), distinct port numbers, or separate hostnames.	`BLUE_PORT`, `GREEN_PORT`, Configuration Management
Environment Isolation	The architectural principle that blue and green environments must be completely independent—deploying to or modifying one environment should never affect the other. This includes separate processes, ports, and configuration.	Dual Environment Setup, Independent Deployment
Expand Phase	The initial stage of the expand-contract migration pattern where new database columns or tables are added in a backward-compatible way (e.g., as nullable columns). Both old and new application versions can work with the expanded schema.	Expand-Contract Pattern, Database Migration, Backward Compatibility
Expand-Contract Pattern	A database migration strategy that maintains backward compatibility by first expanding the schema (adding new nullable columns), then deploying new code that uses the new schema, and finally contracting (removing old columns) only after the new code is stable.	Backward Compatibility, Database Migration, Phased Migration
Feature Flags	Runtime configuration toggles that enable or disable code paths without requiring a deployment. They decouple feature deployment from feature activation, allowing safer gradual rollouts and quick rollbacks via configuration changes.	Canary Analysis, Configuration Management, Gradual Rollout

Term	Definition	Related Concepts
Graceful Reload	Reloading a service's configuration (like a load balancer) without restarting the process, allowing existing connections to complete while new connections use the updated configuration. This enables zero-downtime configuration updates.	Connection Draining, `validate_load_balancer_config()`, Zero-Downtime
Green Environment	The counterpart to the blue environment, typically configured to run on port `GREEN_PORT` (8081). During normal operation, one environment is active (serving traffic) while the other is inactive (idle or running a new version for testing).	`GREEN_PORT`, Environment Identity, Dual Environment
Health Check	An endpoint or probe that verifies an environment's readiness to serve traffic by checking critical dependencies (database, external services) and application health. The load balancer uses health checks to determine which environment receives traffic.	`check_health()`, Readiness Probe, Load Balancer
Idempotent	A property where running the same operation multiple times yields the same result as running it once. Deployment scripts, database migrations, and traffic switches should be idempotent to allow safe retries and prevent inconsistent states.	`apply_migrations()`, `switch_traffic()`, Safe Retries
Immutable Infrastructure	An infrastructure paradigm where servers and environments are never modified in-place after deployment. Instead, new versions are deployed as entirely new, versioned artifacts, and old ones are discarded. This ensures consistency and reproducibility.	Artifact, Versioning, Environment Isolation
Inactive Environment	The production environment not currently receiving live user traffic. This environment can be used for deploying and testing new versions without affecting users. Also called the "staging" or "standby" environment in the blue-green context.	`get_inactive_environment()`, Deployment Target, Smoke Testing
Infrastructure as Code	Managing infrastructure (servers, networks, load balancers) through machine-readable definition files rather than manual configuration. This enables version control, reproducibility, and automated provisioning of blue and green environments.	Environment Isolation, Configuration Management, Automation
Instant Rollback	The ability to revert to a previous application version within seconds by switching traffic back to the still-running previous environment. This is a key advantage of blue-green deployment over traditional strategies.	`rollback_deployment()`, Zero-Downtime, Backward Compatibility
Load Balancer	A reverse proxy (like Nginx or HAProxy) that distributes incoming user requests to backend servers. In blue-green deployment, it routes 100% of traffic to the active environment and can atomically switch traffic between blue and green.	Reverse Proxy, Traffic Switch, `switch_traffic()`
Multi-Region Deployment	Geographically distributing application deployments across multiple cloud regions or data centers for disaster recovery, reduced latency, and increased availability. Blue-green can be extended to multi-region with more complex traffic switching.	Disaster Recovery, Geographic Distribution, High Availability
Phased Migration	Another term for the expand-contract pattern, emphasizing that database schema changes are applied in multiple phases to maintain backward compatibility throughout the deployment process.	Expand-Contract Pattern, Database Migration, Backward Compatibility
Reverse Proxy	A server that sits between clients and backend servers, forwarding client requests to the appropriate backend. Load balancers in blue-green deployments typically function as reverse proxies.	Load Balancer, Traffic Routing, `generate_config_for_env()`
Rollback	Reverting to a previous, known-good application version when a new deployment fails validation or causes issues. In blue-green deployment, rollback is achieved by switching traffic back to the previous environment.	`rollback_deployment()`, Instant Rollback, Failure Recovery
Rolling Update	A deployment strategy where new versions are gradually deployed across instances in a single environment, with the load balancer routing traffic only to healthy instances. Unlike blue-green, this shares infrastructure between old and new versions.	Incremental Deployment, Single Environment, Health Checks
Schema Version	A unique identifier (often a timestamp or sequential number) for a database migration, tracked in a version table like `schema_migrations` to ensure migrations are applied exactly once and in correct order.	`get_current_schema_version()`, `VERSION_TABLE`, Database Migration
Smoke Test	A lightweight validation of critical application functionality after deployment to verify the new version is operational before switching traffic. Smoke tests check API endpoints, database connectivity, and basic user journeys.	`run_smoke_tests()`, Deployment Validation, Pre-Switch Check
Traffic Switch	The atomic redirection of user traffic from the currently active environment to the previously inactive environment. This is the culmination of a successful blue-green deployment, making the new version live.	`switch_traffic()`, Atomic Switch, Load Balancer Controller
Version Table	A database table (typically named `schema_migrations` or similar) that tracks which migrations have been applied, ensuring idempotent application and preventing duplicate	`VERSION_TABLE`, `schema_migrations`, Database Migration Tracking

Term	Definition	Related Concepts
	execution.	
Zero-Downtime	The property of a deployment strategy where users experience no service interruption during application updates. Blue-green deployments achieve zero-downtime by switching traffic between pre-deployed, running environments.	Instant Rollback, Connection Draining, Graceful Reload

Implementation Guidance

While a glossary doesn't require functional implementation, maintaining a living terminology reference is valuable for team alignment. Here's a simple approach to integrate glossary awareness into your deployment scripts.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Terminology Validation	Inline comments in scripts explaining key terms	Automated documentation generation from code comments
Team Reference	Markdown file in project repository	Interactive web-based glossary with search

Recommended File Structure:

Add a documentation directory to your project structure:

```
project-root/
  docs/
    glossary.md                   # This glossary document
    architecture-decisions.md     # ADRs and design rationale
  scripts/
    deploy.sh                     # Main deployment script
    health-check.sh               # Health check utilities
  app/
    # Application code
  migrations/
    # Database migration scripts
```

Core Logic Skeleton Code:

While there's no code to implement for the glossary itself, you can add terminology validation to your deployment scripts to ensure team members understand critical concepts:

```bash
#!/bin/bash

# scripts/deploy.sh - Deployment Orchestrator with terminology reminders


# Display key terminology at start of deployment for clarity

echo "=== Blue-Green Deployment Terminology Reminder ==="

echo "Active Environment: Currently serving live traffic"

echo "Inactive Environment: Target for new deployment"

echo "Traffic Switch: Atomic redirection from active to inactive"

echo "Smoke Test: Validation before switching traffic"

echo "Rollback: Reverting to previous environment on failure"

echo "================================================="


# TODO 1: Call get_current_active_env() to determine active environment

# TODO 2: Call get_inactive_environment() to determine deployment target

# TODO 3: Build artifact (Docker image, binary, etc.)

# TODO 4: Call deploy_to_environment(inactive_env, new_version)

# TODO 5: Call check_health(inactive_env) to verify deployment health

# TODO 6: Call run_smoke_tests(inactive_env, port) to validate functionality

# TODO 7: If smoke tests pass, call switch_traffic(inactive_env)

# TODO 8: If any step fails, call rollback_deployment(inactive_env)
```

Language-Specific Hints:

- **Bash:** Use comments liberally to explain terminology near complex operations
- **Documentation:** Consider using tools like `mkdocs` or `readthedocs` to publish your glossary as part of project documentation
- **Code Reviews:** Include terminology accuracy as a check item in code review checklists

Milestone Checkpoint:

After implementing each milestone, verify that your team understands the relevant terminology:

1. **After Milestone 1:** Team should correctly distinguish between "blue environment" and "green environment" and understand "environment isolation"
2. **After Milestone 2:** Team should understand "traffic switch," "graceful reload," and "connection draining"
3. **After Milestone 3:** Team should understand "smoke test," "deployment orchestration," and "idempotent operations"
4. **After Milestone 4:** Team should understand "expand-contract pattern," "backward compatibility," and "instant rollback"

Debugging Tips:

When troubleshooting deployment issues, verify terminology understanding:

Symptom	Likely Terminology Misunderstanding	Clarification Needed
"Restarted the load balancer and lost connections"	Confusing "restart" with "graceful reload"	Explain that restart terminates connections, while reload preserves them
"Rollback failed because old code couldn't read the database"	Not understanding "backward compatibility"	Clarify that migrations must work with both old and new code versions
"Switched traffic but some users still see old version"	Not understanding "connection draining"	Explain that existing connections complete on old environment before new connections go to new environment