CI/CD Infrastructure Report

Project: Automaspec
Version: 0.1.0

Table of Contents

Executive Summary

The Automaspec project utilizes a comprehensive CI/CD infrastructure built on GitHub Actions, integrated with Vercel for deployment and Lefthook for local git hooks. The infrastructure ensures code quality through automated testing, linting, formatting, and type checking before deployment.

Key Features

- **Automated Quality Checks**: Formatting, linting, type checking, and database schema validation
- **Multi-Environment Deployment**: Preview and production deployments via Vercel
- **Test Result Synchronization**: Automated sync of test results to Automaspec platform
- **Local Development Support**: Manual workflow for local development testing
- **Git Hooks**: Pre-commit and pre-push hooks for early error detection

---

CI/CD Pipeline Architecture

The Automaspec CI/CD pipeline follows industry best practices with a clear separation between Continuous Integration (CI) and Continuous Deployment (CD) stages. The pipeline is designed to ensure code quality, reliability, and automated deployment.

CI Stages (Continuous Integration)

The CI pipeline consists of **3 mandatory stages** plus additional quality checks:

Stage 1: Code Quality Checks **Purpose:** Ensure code consistency and catch issues early before build

Components: - **Linting (oxlint)**: Static code analysis to detect potential bugs, code smells, and enforce coding standards - **Formatting (oxfmt)**: Automatic code formatting to maintain consistent style across the codebase - **Database Schema Validation**: Validates database schema changes using Drizzle Kit to prevent migration errors

Why Automated: - Prevents inconsistent code style from entering the repository - Catches syntax errors and potential bugs before compilation - Ensures database migrations are valid before deployment - Reduces code review time by handling formatting automatically

Implementation: - Runs in pre-commit hooks (via Lefthook) for immediate feedback - Also runs in CI pipeline to catch any bypassed hooks - Auto-fixes formatting and linting issues where possible

Stage 2: Build **Purpose:** Compile and build the application artifacts

Components: - **Dependency Installation**: Install project dependencies using pnpm with frozen lockfile - **Next.js Build**: Compile TypeScript, bundle assets, optimize production build - **Type Checking**: Verify TypeScript types are correct (`next typegen && tsc --noEmit`)

Why Automated: - Ensures the application can be built successfully - Catches compilation errors early in the pipeline - Validates that all dependencies are correctly resolved - Produces optimized production-ready artifacts

Implementation: - Uses `pnpm install --frozen-lockfile` for reproducible builds - Next.js build process handles bundling, optimization, and static generation - Type checking runs in parallel with tests for faster feedback

Stage 3: Testing **Purpose:** Validate application functionality and prevent regressions

Components: - **Unit Tests**: Component and function-level tests using Vitest - **Integration Tests**: End-to-end workflow tests - **Database Tests**: Schema and ORM validation tests - **Test Coverage**: Coverage reporting via @vitest/coverage-v8

Why Automated: - Ensures new changes don't break existing functionality - Validates business logic correctness - Provides confidence for deployments - Generates test reports for quality tracking

Implementation: - Runs all tests in CI pipeline (`pnpm test run`) - Test results exported as JSON for integration with Automaspec platform - Tests run in parallel where possible for faster execution - Coverage reports generated for quality metrics

CD Stages (Continuous Deployment)

The CD pipeline includes automated deployment to multiple environments:

Deployment to Preview Environment **Purpose:** Provide preview deployments for pull requests and feature branches

Implementation: - Automatic deployment after successful CI checks - Uses Vercel preview environment - Accessible via unique preview URLs - Allows stakeholders to review changes before merge

Deployment to Production Environment **Purpose:** Deploy validated code to production

Implementation: - Only triggers on `main` branch - Requires all CI checks to pass - Uses Vercel production environment - Includes production-specific optimizations

Why Automated: - Reduces manual deployment errors - Ensures consistent deployment process - Enables rapid iteration and rollback capabilities - Eliminates "works on my machine" issues

Pipeline Flow Diagram

See `docs/ci-cd-infrastructure-diagram.svg` for a detailed visual representation of the pipeline architecture, workflow dependencies, and execution flow.

GitHub Actions Workflows

The project contains three GitHub Actions workflows located in `.github/workflows/`:

1. CI/CD Pipeline (`ci-cd.yml`)

Purpose: Main continuous integration and deployment pipeline

Triggers: - Push to `main`, `master`, or `dev` branches - Pull requests to `main`, `master`, or `dev` branches

Jobs:

Quality Checks

- **Runner:** `ubuntu-latest`
- **Steps:**
  1. Checkout repository
  2. Install pnpm package manager
  3. Setup Node.js version 22 with pnpm cache
  4. Install dependencies with frozen lockfile
  5. Run pre-commit hooks (format, lint, db-check)

6. Run pre-push hooks (typecheck, tests)

Deploy Preview

- **Dependencies:** Requires `quality-checks` job to succeed
- **Runner:** `ubuntu-latest`
- **Steps:**
  1. Checkout repository
  2. Install Vercel CLI globally
  3. Pull Vercel environment information (preview)
  4. Build project artifacts
  5. Deploy to Vercel preview environment

Deploy Production

- **Dependencies:** Requires `quality-checks` job to succeed
- **Condition:** Only runs on `main` branch (`github.ref == 'refs/heads/main'`)
- **Runner:** `ubuntu-latest`
- **Steps:**
  1. Checkout repository
  2. Install Vercel CLI globally
  3. Pull Vercel environment information (production)
  4. Build project artifacts with production flag
  5. Deploy to Vercel production environment

Environment Variables: - `VERCEL_ORG_ID`: From GitHub secrets - `VERCEL_PROJECT_ID`: From GitHub secrets - `VERCEL_TOKEN`: From GitHub secrets (used in deploy steps)

---

2. Sync Test Results to Automaspec (`automaspec-sync.yml`)

Purpose: Automatically sync test results to the Automaspec platform after test execution

Triggers: - Push to `main`, `master`, or `dev` branches - Pull requests to `main`, `master`, or `dev` branches - Manual workflow dispatch

Jobs:

Test and Sync

- **Runner:** `ubuntu-latest`
- **Steps:**
  1. Checkout repository
  2. Install pnpm package manager
  3. Setup Node.js version 22 with pnpm cache
  4. Install dependencies with frozen lockfile
  5. Run tests with JSON reporter (outputs to `test-results.json`)
     - Continues on error to ensure sync happens even if tests fail
  6. Sync test results to Automaspec via webhook
     - Uses `AUTOMASPEC_API_KEY` secret for authentication
     - Sends JSON payload to `AUTOMASPEC_WEBHOOK_URL`
     - Runs even if tests fail (`if: always()`)

Secrets Required: - `AUTOMASPEC_API_KEY`: API key for webhook authentication - `AUTOMASPEC_WEBHOOK_URL`: Endpoint URL for test result synchronization

---

3. Local Dev - Sync Test Results (`local-sync.yml`)

Purpose: Manual workflow for syncing test results to a local development instance

Triggers: - Manual workflow dispatch only

Inputs: - `base_url`: Base URL of the Automaspec instance (default: `http://localhost:3000`) - `api_key`: API key for authentication (required)

Jobs:

Sync Local

- **Runner:** `ubuntu-latest`
- **Steps:**
  1. Checkout repository
  2. Install pnpm package manager
  3. Setup Node.js version 22 with pnpm cache
  4. Install dependencies (without frozen lockfile for local dev)
  5. Run tests with JSON reporter (outputs to `test-results.json`)
     - Continues on error
  6. Display test results (first 100 lines)
  7. Sync test results to local Automaspec instance
     - Uses provided `api_key` input
     - Posts to `{base_url}/api/webhook/sync-tests`
     - Includes error handling for server availability

Pre-commit and Pre-push Hooks

The project uses **Lefthook** for git hook management, configured in `lefthook.yml`.

Pre-commit Hooks

Executed automatically before commits:

1. **Database Schema Check**
   - **Trigger:** Changes to `db/schema/*.ts` files
   - **Command:** `pnpm dbc` (drizzle-kit check)
   - **Purpose:** Validates database schema integrity
2. **Format**
   - **Trigger:** Changes to `*.{ts,tsx,mts}` files
   - **Command:** `pnpm format {staged_files}` (oxfmt)
   - **Behavior:** Auto-stages fixed files
   - **Purpose:** Ensures consistent code formatting
3. **Lint**
   - **Trigger:** Changes to `*.{ts,tsx,mts}` files
   - **Command:** `pnpm lint {staged_files}` (oxlint)
   - **Behavior:** Auto-stages fixed files
   - **Purpose:** Catches code quality issues early

Pre-push Hooks

Executed automatically before pushes (runs in parallel):

1. **Type Check**
   - **Trigger:** Changes to `*.{ts,tsx,mts}` files
   - **Command:** `pnpm typecheck` (next typegen && tsc –noEmit)
   - **Purpose:** Ensures TypeScript type safety
2. **Tests**
   - **Trigger:** Changes to `*.{ts,tsx,mts}` files
   - **Command:** `pnpm test run` (vitest)
   - **Purpose:** Validates code functionality

What is Automated and Why

Code Quality Automation

What: Formatting, linting, and database schema validation
Why: - Ensures consistent code style without manual intervention - Catches errors before they reach code review - Reduces cognitive load on developers - Prevents database migration issues in production

Build Automation

What: Dependency installation, TypeScript compilation, Next.js build
Why: - Ensures reproducible builds across environments - Catches build errors early - Validates that code compiles correctly - Produces optimized production artifacts

Testing Automation

What: Unit tests, integration tests, test coverage reporting
Why: - Prevents regressions from reaching production - Validates business logic correctness - Provides confidence for deployments - Enables test-driven development practices

Deployment Automation

What: Automatic deployment to preview and production environments
Why: - Eliminates manual deployment errors - Enables rapid iteration cycles - Ensures consistent deployment process - Reduces deployment time from hours to minutes

Test Result Synchronization

What: Automatic sync of test results to Automaspec platform
Why: - Provides centralized test result tracking - Enables historical test analysis - Supports test specification management - Facilitates quality metrics reporting

---

Artifacts

The CI/CD pipeline generates the following artifacts:

Build Artifacts

1. **Next.js Production Build**
   - **Location:** `.next/` directory (generated during build)
   - **Contents:**
     - Compiled JavaScript bundles
     - Optimized CSS files
     - Static HTML pages (where applicable)
     - Server-side rendering artifacts
   - **Usage:** Deployed to Vercel for production/preview environments
   - **Size:** Optimized and minified for production
2. **TypeScript Type Definitions**
   - **Location:** Generated during `next typegen`
   - **Contents:** Type definitions for API routes and server components
   - **Usage:** Used for type checking and IDE support

Test Artifacts

3. **Test Results JSON**
   - **Location:** `test-results.json`
   - **Contents:**
     - Test execution results
     - Test names and statuses
     - Execution times
     - Error messages (if any)
   - **Usage:**
     - Synced to Automaspec platform via webhook
     - Used for test tracking and reporting
   - **Format:** Vitest JSON reporter format
4. **Test Coverage Reports**
   - **Location:** `coverage/` directory (when coverage is enabled)
   - **Contents:**
     - Line coverage percentages
     - Branch coverage data

  - Function coverage metrics
- **Usage:** Quality metrics and coverage tracking

Deployment Artifacts

5. **Vercel Deployment Artifacts**
   - **Location:** Vercel platform (pre-built artifacts)
   - **Contents:** Pre-built Next.js application ready for deployment
   - **Usage:** Deployed to preview/production environments
   - **Deployment Method:** `vercel deploy --prebuilt`

Docker Artifacts (Optional)

6. **Docker Images**
   - **Location:** Local Docker registry (when using Docker Compose)
   - **Contents:** Containerized application image
   - **Usage:** Local development and containerized deployments
   - **Build Command:** `docker compose up --build`

Artifact Retention

- **GitHub Actions:** Artifacts retained according to GitHub Actions retention policies
- **Vercel:** Build artifacts cached and reused when possible
- **Test Results:** Synced to Automaspec platform for long-term storage

---

Environments

The project maintains three distinct environments, each serving a specific purpose in the development lifecycle:

1. Development Environment (Local)

Purpose: Local development and testing

Characteristics: - Runs on developer's machine - Uses local database (Turso dev server with `db/local.db`) - Hot module replacement enabled (Turbopack) - Development dependencies available - No production optimizations

Access: - Local URL: `http://localhost:3000` - Database: Local SQLite file via Turso dev server - Environment Variables: `.env.local` file

Usage: - Active development - Feature implementation - Local testing and debugging - Manual test execution

Deployment: - Manual start via `pnpm dev` - No CI/CD deployment - Developer-controlled

2. Preview Environment (Vercel Preview)

Purpose: Review and validation of changes before merge

Characteristics: - Automatically deployed on every push/PR - Uses preview Vercel configuration - Production-like optimizations - Isolated from production data - Temporary (deleted after PR merge/close)

Access: - Unique preview URL per deployment (e.g., `automaspec-xyz.vercel.app`) - Generated automatically by Vercel - Accessible to team members and stakeholders

Usage: - Code review validation - Stakeholder approval - Integration testing - Visual regression testing

Deployment: - Automatic via GitHub Actions - Triggered on push to any branch or PR - Requires successful CI checks

Configuration: - Environment variables: Vercel Preview environment - Database: Shared test database or preview-specific instance - Build: Production build with preview optimizations

3. Production Environment (Vercel Production)

Purpose: Live application serving end users

Characteristics: - Production-grade optimizations - CDN distribution - Production database - Monitoring and analytics enabled - High availability

Access: - Production URL: Configured in Vercel project settings - Publicly accessible (with authentication) - SSL/TLS enabled

Usage: - Live application for end users - Production data management - Real-world usage patterns

Deployment: - Automatic via GitHub Actions - Only triggers on `main` branch - Requires all CI checks to pass - Includes production-specific build flags (`--prod`)

Configuration: - Environment variables: Vercel Production environment - Database: Production database instance - Build: Full production optimizations - Monitoring: Vercel Analytics and error tracking

Environment Separation

Data Isolation: - Each environment uses separate database instances - No data sharing between environments - Preview environment uses test data

Configuration Management: - Environment-specific variables in Vercel - No hardcoded environment values - Secrets managed via GitHub Secrets and Vercel

Deployment Isolation: - Preview deployments don't affect production - Production deployments require explicit approval (via branch protection) - Rollback capabilities available in Vercel

---

Deployment Strategy

Platform: Vercel

The project uses Vercel for hosting and deployment with the following configuration:

- **Project ID:** `prj_xP6puzyfVhHwlCniCsqQ9pFYbVVA`
- **Organization ID:** `team_sXUvzoND1TcTxUAjfyt5n8bY`

Deployment Environments

1. **Preview Environment**
   - Deployed on every push/PR (after quality checks pass)
   - Uses preview configuration
   - Accessible via Vercel preview URLs
2. **Production Environment**
   - Deployed only from `main` branch
   - Uses production configuration (`--prod` flag)
   - Requires successful quality checks

Deployment Process

1. Quality checks must pass before deployment
2. Vercel CLI pulls environment-specific configuration
3. Project artifacts are built using Vercel build system
4. Pre-built artifacts are deployed to Vercel

Quality and Error Handling

Pipeline Failure Handling

The CI/CD pipeline is designed to fail fast and provide clear feedback:

Build Failures Behavior: - Pipeline immediately stops on build errors - No deployment occurs if build fails - Clear error messages displayed in GitHub Actions logs

Error Sources: - TypeScript compilation errors - Missing dependencies - Build configuration issues - Next.js build errors

Handling: - Build errors are caught during the "Quality Checks" job - Detailed error logs available in GitHub Actions - Developers receive notifications via GitHub

Test Failures **Behavior:** - Pipeline fails if tests fail (except in test sync workflow) - Test results are still synced to Automaspec even on failure - Detailed test failure reports available

Error Sources: - Unit test failures - Integration test failures - Test timeout errors - Test environment setup issues

Handling: - Test failures prevent deployment - Test results JSON includes failure details - Coverage reports show which areas lack tests

Deployment Failures **Behavior:** - Deployment job fails if deployment errors occur - Previous successful deployment remains active - Rollback capabilities available in Vercel

Error Sources: - Vercel API errors - Environment variable misconfiguration - Build artifact issues - Network connectivity problems

Handling: - Deployment errors logged in GitHub Actions - Vercel dashboard shows deployment status - Manual rollback available via Vercel CLI or dashboard

Logging and Reporting

CI/CD Logs **Location:** GitHub Actions workflow logs

Contents: - Step-by-step execution logs - Build output - Test execution details - Error messages and stack traces - Deployment status

Access: - Available in GitHub Actions UI - Downloadable as logs archive - Searchable and filterable

Test Reports **Test Count:** - Total number of tests executed - Passed/failed/skipped counts - Displayed in GitHub Actions summary

Linter Results: - oxlint output shows warnings and errors - Auto-fixable issues are automatically resolved - Remaining issues displayed in logs

Coverage Reports: - Available when coverage is enabled - Shows line, branch, and function coverage - Identifies untested code areas

Error Recovery

Automatic Recovery

- **Formatting/Linting:** Auto-fixed issues are staged automatically
- **Dependency Issues:** Clear error messages guide resolution
- **Build Cache:** Cached dependencies speed up retries

Manual Recovery

- **Failed Builds:** Fix errors and push again
- **Failed Tests:** Fix test issues and re-run pipeline
- **Failed Deployments:** Use Vercel dashboard to retry or rollback

Quality Gates

Pre-Deployment Checks: 1. All linting checks pass 2. Code formatting is correct 3. TypeScript compilation succeeds 4. All tests pass 5. Database schema is valid

Deployment Requirements: - Quality checks must pass before deployment - Production deployment requires `main` branch - Preview deployment available for all branches

Post-Deployment Validation: - Vercel provides deployment status - Health checks available via application endpoints - Monitoring tracks deployment success

---

Security

Secrets Management

GitHub Secrets All sensitive information is stored securely in GitHub Secrets, never in code or configuration files:

CI/CD Pipeline Secrets: - `VERCEL_ORG_ID`: Vercel organization identifier - `VERCEL_PROJECT_ID`: Vercel project identifier - `VERCEL_TOKEN`: Vercel authentication token

Test Sync Secrets: - `AUTOMASPEC_API_KEY`: API key for webhook authentication - `AUTOMASPEC_WEBHOOK_URL`: Webhook endpoint URL

Security Practices: - No secrets in repository files - No secrets in workflow YAML files (only references) - Secrets are encrypted at rest - Access controlled via GitHub permissions - Secrets are masked in logs

Vercel Environment Variables **Production Environment:** - Database connection strings - API keys for external services - Authentication secrets - Feature flags

Preview Environment: - Separate test database credentials - Test API keys - Preview-specific configuration

Security Practices: - Environment variables encrypted in Vercel - Separate variables for each environment - No secrets exposed in build logs - Access controlled via Vercel team permissions

Static Security Analysis

Dependency Vulnerability Scanning **Tool:** pnpm (via `pnpm audit` or similar)

Implementation: - Dependency vulnerabilities checked during development - `node-modules-inspector` available for dependency analysis - Regular dependency updates via `taze` tool

Coverage: - All npm/pnpm dependencies scanned - Known vulnerabilities reported - Update recommendations provided

Future Enhancements: - Automated `npm audit` in CI pipeline - Dependabot integration for automated updates - SonarCloud integration (optional)

Code Security Practices **Current Practices:** - No hardcoded secrets in code - Environment variables for configuration - TypeScript for type safety - Input validation via Zod schemas - SQL injection prevention via Drizzle ORM

Security Considerations: - Database queries use parameterized statements (Drizzle ORM) - Authentication handled by Better Auth library - API routes validate input schemas - CORS configured appropriately

Security Best Practices Implemented

1. **Secrets Storage:**
   - All secrets in GitHub Secrets or Vercel environment variables
   - No secrets committed to repository
   - Secrets rotated regularly
2. **Access Control:**
   - GitHub Actions workflows require repository access
   - Vercel deployments require team membership
   - API endpoints protected with authentication
3. **Dependency Management:**
   - Frozen lockfile ensures reproducible builds
   - Regular dependency updates
   - Vulnerability scanning available
4. **Build Security:**
   - Non-root user in Docker containers
   - Minimal base images (Alpine Linux)
   - No unnecessary tools in production images

Testing Infrastructure

Test Framework: Vitest

- **Test Runner:** Vitest 4.0.16
- **Test Environment:** jsdom (for React component testing)
- **Coverage Tool:** @vitest/coverage-v8

Test Scripts

- `pnpm test`: Run tests in watch mode
- `pnpm test:coverage`: Run tests with coverage report
- `pnpm test run`: Run tests once (used in CI)

Test Structure

Tests are organized in `__tests__/` directory: - Component tests: `__tests__/components/` - Database tests: `__tests__/db/` - Integration tests: `__tests__/integration/` - Library tests: `__tests__/lib/` - ORPC route tests: `__tests__/orpc/routes/`

Test Reporting

- JSON reporter used for CI/CD integration
- Output file: `test-results.json`
- Results automatically synced to Automaspec platform

Dependencies and Tools

Package Manager

- **pnpm** 10.26.0 (specified in `packageManager` field)
- Uses frozen lockfile in CI (`--frozen-lockfile`)

Node.js Version

- **Node.js 22** (specified in all workflows)

Code Quality Tools

- **oxfmt** 0.14.0: Code formatter
- **oxlint** 1.33.0: Fast linter
- **TypeScript** 5.9.3: Type checking
- **Lefthook** 2.0.12: Git hooks manager

Database Tools

- **Drizzle ORM** 0.44.7: Database ORM
- **Drizzle Kit** 0.31.8: Database migrations and schema management
- **Turso**: Database hosting (local dev via `db:local`)

Build Tools

- **Next.js** 16.1.0: React framework
- **Vercel CLI**: Deployment tool

Testing Tools

- **Vitest** 4.0.16: Test framework
- **React Testing Library** 16.3.1: Component testing
- **jsdom** 26.1.0: DOM environment for tests

Secrets and Environment Variables

GitHub Secrets Required

CI/CD Pipeline (`ci-cd.yml`)

- `VERCEL_ORG_ID`: Vercel organization identifier
- `VERCEL_PROJECT_ID`: Vercel project identifier
- `VERCEL_TOKEN`: Vercel authentication token

Test Sync Workflow (`automaspec-sync.yml`)

- `AUTOMASPEC_API_KEY`: API key for webhook authentication
- `AUTOMASPEC_WEBHOOK_URL`: Endpoint URL for test synchronization

Environment Variables

- `VERCEL_ORG_ID`: Set at workflow level in `ci-cd.yml`
- `VERCEL_PROJECT_ID`: Set at workflow level in `ci-cd.yml`

Workflow Triggers

Automatic Triggers

1. **Push Events**
   - Branches: `main`, `master`, `dev`
   - Triggers: CI/CD pipeline, test sync workflow
   - **Rationale:** Ensures all code changes are validated before merge
2. **Pull Request Events**
   - Target branches: `main`, `master`, `dev`
   - Triggers: CI/CD pipeline, test sync workflow
   - **Rationale:** Validates changes before merge, provides preview deployments

Manual Triggers

1. **Workflow Dispatch**
   - Available for: `automaspec-sync.yml`, `local-sync.yml`
   - `local-sync.yml` requires user inputs (base_url, api_key)
   - **Rationale:** Allows on-demand test synchronization and local testing

---

Implementation Details

Dependency Caching

pnpm Cache: - **Implementation:** GitHub Actions `cache: 'pnpm'` in Node.js setup - **Location:** Cached in GitHub Actions runner - **Benefits:** - Faster dependency installation - Reduced network usage - Consistent dependency resolution

Cache Strategy: - Cache key based on `pnpm-lock.yaml` hash - Cache restored before dependency installation - Cache saved after successful installation

Cache Invalidation: - Automatically invalidated when lockfile changes - Manual cache clearing available in GitHub Actions - Cache persists across workflow runs

Artifact Generation

Build Artifacts: - Generated during `vercel build` step - Stored temporarily in GitHub Actions runner - Uploaded to Vercel for deployment

Test Artifacts: - `test-results.json` generated by Vitest - Stored in workflow runner - Uploaded to Automaspec via webhook

Artifact Retention: - GitHub Actions: Default retention (90 days for public repos) - Vercel: Build artifacts cached and reused - Automaspec: Test results stored long-term

Deployment Forms

Primary Deployment: Vercel - **Type:** Cloud platform deployment - **Method:** Pre-built artifacts via Vercel CLI - **Environments:** Preview and Production - **Benefits:** - Automatic SSL/TLS - Global CDN distribution - Serverless function support - Built-in analytics

Secondary Deployment: Docker Compose - **Type:** Containerized deployment - **Method:** Docker Compose for local development - **Usage:** Local testing and development - **Benefits:** - Consistent local environment - Easy environment replication - Production-like local setup

Pipeline Execution

Parallel Execution: - Pre-push hooks run in parallel (typecheck and tests) - Multiple jobs can run simultaneously when independent - Reduces total pipeline execution time

Sequential Execution: - Quality checks must complete before deployment - Deployment jobs depend on quality checks - Ensures only validated code is deployed

Conditional Execution: - Production deployment only on `main` branch - Preview deployment for all branches - Test sync runs even if tests fail (`if: always()`)

Error Handling in Pipeline

Fail-Fast Strategy: - Pipeline stops immediately on errors - No partial deployments - Clear error messages for debugging

Error Notifications: - GitHub Actions provides email notifications - Workflow status badges available - Integration with GitHub PR status checks

Recovery Process: 1. Developer fixes the issue 2. Pushes changes to trigger new pipeline 3. Pipeline re-runs all checks 4. Deployment proceeds if checks pass

---

Workflow Triggers

Automatic Triggers

1. **Push Events**
   - Branches: `main`, `master`, `dev`
   - Triggers: CI/CD pipeline, test sync workflow
2. **Pull Request Events**
   - Target branches: `main`, `master`, `dev`
   - Triggers: CI/CD pipeline, test sync workflow

Manual Triggers

1. **Workflow Dispatch**
   - Available for: `automaspec-sync.yml`, `local-sync.yml`
   - `local-sync.yml` requires user inputs (base_url, api_key)

Workflow Dependencies

The CI/CD pipeline follows a dependency-based execution model:

Quality Checks → **Deploy Preview** (for all branches)
Quality Checks → **Deploy Production** (for main branch only)
Test Sync → Runs independently (parallel to deployment)

See `docs/ci-cd-infrastructure-diagram.svg` for a detailed visual representation of workflow dependencies and execution flow.

Best Practices Implemented

1. **Frozen Lockfile**: Ensures consistent dependency versions in CI
2. **Parallel Execution**: Pre-push hooks run in parallel for faster feedback
3. **Auto-fix**: Pre-commit hooks auto-stage fixed files
4. **Fail-fast**: Quality checks must pass before deployment
5. **Environment Separation**: Preview and production deployments are isolated
6. **Test Continuity**: Test results sync even if tests fail
7. **Local Development**: Manual workflow supports local testing
8. **Type Safety**: TypeScript checking enforced before push
9. **Database Validation**: Schema changes validated before commit
10. **Code Formatting**: Consistent code style enforced automatically

Conclusion

The Automaspec CI/CD infrastructure provides a robust foundation for automated testing, code quality enforcement, and deployment. The combination of GitHub Actions, Lefthook, and Vercel ensures that code changes are validated at multiple stages before reaching production, maintaining high code quality and reliability.

The infrastructure supports both automated workflows for production deployments and manual workflows for local development, providing flexibility for different development scenarios.