Repository Analysis Logic for DevOps Questions

1. Application Type & Scale Detection

Analysis Algorithm:

```
ANALYZE file_structure AND package_files:
  IF (has_frontend_files AND has_backend_files):
    return "Web Application (Frontend + Backend)"
  ELIF (has_api_routes AND no_frontend_files):
    return "API Service (Backend only)"
  ELIF (single_codebase AND mixed_concerns):
    return "Full-Stack Monolith"
  ELIF (multiple_services_detected OR docker_compose_services > 1):
    return "Microservices Architecture"
```

Detection Logic:

- **Frontend files**: `src/components/`, `public/`, `index.html`, React/Vue/Angular configs
- **Backend files**: `routes/`, `controllers/`, `models/`, API frameworks
- **Multiple services**: Multiple `Dockerfile`s, `docker-compose.yml` with >1 service, separate `package.json` files
- **Monolith indicators**: Single entry point, mixed frontend/backend in same directory structure

LLM Prompt:

```
Analyze this repository structure and determine the application architecture type:

Repository files: {file_tree}
Package files: {package_json_content}
Framework indicators: {detected_frameworks}

Classify as one of:
1. Web Application (Frontend + Backend)
2. API Service (Backend only)
3. Full-Stack Monolith
4. Microservices Architecture

Provide reasoning based on folder structure, dependencies, and entry points.
```

---

3. Database Requirements Detection

Analysis Algorithm:

```
SCAN dependencies AND environment_files AND config_files:
  database_indicators = []

  FOR each dependency:
    IF dependency IN sql_libs: database_indicators.append("SQL")
    IF dependency IN nosql_libs: database_indicators.append("NoSQL")
    IF dependency IN cache_libs: database_indicators.append("Cache")
    IF dependency IN inmemory_libs: database_indicators.append("In-Memory")

  SCAN environment_files FOR:
    database_urls, connection_strings, db_host_variables

  SCAN code_files FOR:
    database_connection_patterns, ORM_usage, query_patterns

  return most_likely_database_type(database_indicators)
```

Detection Patterns:

- **SQL indicators**: `pg`, `mysql2`, `sequelize`, `prisma`, `typeorm`, `django.db`
- **NoSQL indicators**: `mongodb`, `mongoose`, `pymongo`, `dynamodb`
- **Cache indicators**: `redis`, `ioredis`, `node_redis`, `redis-py`
- **Environment variables**: `DATABASE_URL`, `DB_HOST`, `REDIS_URL`, `MONGO_URI`
- **Config files**: `database.yml`, `knexfile.js`, `prisma/schema.prisma`

LLM Prompt:

```
Analyze this repository's database requirements:

Dependencies: {package_dependencies}
Environment variables: {env_file_contents}
Database-related files: {db_config_files}
Code patterns: {database_code_snippets}

Determine the database requirements:
1. No Database
2. SQL Database (specify: PostgreSQL/MySQL)
3. NoSQL Database (specify: MongoDB/DynamoDB)
4. Cache Layer (Redis)
5. In-Memory Database

Provide confidence level and reasoning.
```

---

7. Cluster Architecture Requirements

Analysis Algorithm:

```
ANALYZE codebase_complexity AND traffic_patterns AND availability_requirements:
  complexity_score = calculate_complexity(
    lines_of_code,
    number_of_services,
    dependency_count,
    team_size_indicators
  )

  availability_requirements = detect_availability_needs(
    has_health_checks,
    has_monitoring,
    has_backup_strategies,
    compliance_indicators
  )

  IF (complexity_score > 8 OR availability_requirements == "high"):
    return "Multi-Region"
  ELIF (complexity_score > 5 OR has_load_balancing):
    return "Multi-AZ"
  ELSE:
    return "Single Region"
```

Detection Logic:

- **Complexity indicators**: LOC > 50k, >5 services, >100 dependencies
- **High availability needs**: Health check endpoints, circuit breakers, retry logic
- **Compliance requirements**: SOC2/HIPAA mentions in docs, audit logging
- **Load balancing**: Nginx configs, HAProxy, load balancer references

LLM Prompt:

```
Analyze this repository's infrastructure requirements:

Codebase metrics: {complexity_metrics}
Services detected: {services_count}
High availability patterns: {ha_patterns_found}
Compliance requirements: {compliance_mentions}
Documentation: {readme_content}

Recommend cluster architecture:
1. Single Region - Standard EKS cluster
2. Multi-AZ - Cross-AZ deployment
3. Multi-Region - Regional clusters
4. Hybrid Cloud - On-premises + cloud

Consider complexity, availability needs, and scale requirements.
```

---

8. Networking Requirements Detection

Analysis Algorithm:

```
ANALYZE application_type AND security_patterns AND external_integrations:
  public_indicators = scan_for_patterns([
    "public API", "web application", "frontend", "static assets",
    "cdn", "cloudfront", "public endpoints"
  ])

  private_indicators = scan_for_patterns([
    "internal API", "private network", "VPN", "internal services",
    "database connections", "admin panels"
  ])

  api_gateway_indicators = scan_for_patterns([
    "rate limiting", "authentication", "API versioning",
    "swagger", "openapi", "multiple endpoints"
  ])

  return determine_networking_needs(public_indicators, private_indicators,
api_gateway_indicators)
```

Detection Patterns:

- **Public access**: Frontend frameworks, static assets, public API docs
- **Private access**: Admin routes, internal APIs, database connections
- **CDN needs**: Static assets, image optimization, global user base
- **API Gateway**: Rate limiting, auth middleware, API versioning, Swagger docs

LLM Prompt:

```
Analyze this repository's networking and access requirements:

Application type: {app_type}
Public endpoints: {public_endpoints_found}
Authentication patterns: {auth_patterns}
Static assets: {static_assets_detected}
API documentation: {api_docs_found}
Geographic considerations: {user_base_hints}

Determine networking requirements:
1. Public Internet Access - ALB + public subnets
2. Private with VPN - Private subnets + VPN gateway
3. CDN Required - CloudFront integration
4. API Gateway - AWS API Gateway + service mesh

Consider security, performance, and user access patterns.
```

---

9. Storage Requirements Detection

Analysis Algorithm:

```
ANALYZE file_operations AND data_persistence AND upload_patterns:
  persistent_storage_needs = []

  IF scan_for_file_uploads(): persistent_storage_needs.append("Object Storage")
  IF scan_for_file_operations(): persistent_storage_needs.append("File Storage")
  IF scan_for_database_files(): persistent_storage_needs.append("Block Storage")
  IF scan_for_temporary_files_only(): persistent_storage_needs.append("No Persistent
Storage")

  return prioritize_storage_needs(persistent_storage_needs)
```

Detection Patterns:

- **File uploads**: Multer, express-fileupload, file upload endpoints
- **File operations**: File system operations, file processing, document handling
- **Database files**: SQLite, local database files, data directories
- **Temporary only**: In-memory processing, stateless operations

LLM Prompt:

```
Analyze this repository's storage requirements:

File operations: {file_operations_found}
Upload functionality: {upload_patterns}
Data persistence: {persistence_patterns}
Temporary file usage: {temp_file_patterns}
Database setup: {database_storage_needs}

Determine storage requirements:
1. No Persistent Storage - EmptyDir volumes only
2. File Storage - EFS integration
3. Block Storage - EBS CSI driver
4. Object Storage - S3 integration

Consider data persistence, file handling, and performance needs.
```

---

10. Observability Level Detection

Analysis Algorithm:

```
ANALYZE existing_monitoring AND logging_patterns AND error_handling:
  monitoring_maturity = 0

  IF has_health_endpoints(): monitoring_maturity += 1
  IF has_metrics_collection(): monitoring_maturity += 2
  IF has_structured_logging(): monitoring_maturity += 2
  IF has_error_tracking(): monitoring_maturity += 1
  IF has_performance_monitoring(): monitoring_maturity += 2
  IF has_distributed_tracing(): monitoring_maturity += 3

  return map_maturity_to_observability_level(monitoring_maturity)
```

Detection Patterns:

- **Health endpoints**: `/health`, `/ping`, health check routes
- **Metrics**: Prometheus metrics, StatsD, custom metrics
- **Structured logging**: JSON logs, winston, structured log formats
- **Error tracking**: Sentry, Bugsnag, error handling middleware
- **Performance monitoring**: APM tools, performance middleware
- **Distributed tracing**: OpenTelemetry, Jaeger, Zipkin

LLM Prompt:

```
Analyze this repository's current observability and monitoring setup:

Health endpoints: {health_endpoints_found}
Logging patterns: {logging_setup}
Metrics collection: {metrics_patterns}
Error handling: {error_handling_patterns}
Performance monitoring: {performance_tools}
Tracing setup: {tracing_indicators}

Recommend observability level:
1. Basic Monitoring - CloudWatch + basic dashboards
2. Standard Observability - Prometheus + Grafana + AlertManager
3. Full Observability - ELK/EFK stack + distributed tracing
4. Enterprise Monitoring - DataDog/New Relic integration

Consider current maturity and application complexity.
```

---

11. Logging Strategy Detection

Analysis Algorithm:

```
ANALYZE logging_implementation AND log_volume AND structure:
  current_logging = analyze_logging_setup()

  log_complexity_score = calculate_log_complexity(
    structured_logs_present,
    log_volume_indicators,
    multiple_services,
    compliance_requirements
  )

  return recommend_logging_strategy(current_logging, log_complexity_score)
```

Detection Patterns:

- **Basic logging**: console.log, print statements, basic logging
- **Structured logging**: JSON logs, key-value pairs, consistent format
- **Log volume**: High traffic indicators, batch processing, frequent operations

- **Multiple services**: Microservices, distributed logging needs

LLM Prompt:

```
Analyze this repository's logging requirements:

Current logging: {logging_patterns_found}
Log structure: {log_format_analysis}
Application complexity: {complexity_indicators}
Traffic patterns: {traffic_volume_hints}
Compliance needs: {compliance_requirements}

Recommend logging strategy:
1. Container Logs Only - kubectl logs access
2. Centralized Logging - FluentBit + CloudWatch/S3
3. Log Analytics - ELK stack with Kibana
4. Structured Logging - JSON logs + log parsing

Consider volume, structure, and analysis requirements.
```

---

13. CI/CD Integration Detection

Analysis Algorithm:

```
ANALYZE existing_ci_cd AND repository_platform AND team_preferences:
  current_ci_cd = scan_for_ci_files([
    ".github/workflows/",
    ".gitlab-ci.yml",
    "Jenkinsfile",
    "buildspec.yml"
  ])

  IF current_ci_cd: return enhance_existing_ci_cd(current_ci_cd)

  repository_platform = detect_git_platform()
  return recommend_ci_cd_for_platform(repository_platform)
```

Detection Patterns:

- **GitHub**: `.github/workflows/` directory, GitHub-specific features
- **GitLab**: `.gitlab-ci.yml`, GitLab-specific configurations

- **Jenkins**: `Jenkinsfile`, Jenkins-specific patterns
- **AWS**: `buildspec.yml`, AWS CodeBuild patterns

LLM Prompt:

```
Analyze this repository's CI/CD setup and requirements:

Existing CI/CD files: {ci_cd_files_found}
Repository platform: {git_platform}
Build requirements: {build_patterns}
Testing setup: {test_configurations}
Deployment patterns: {deployment_hints}

Recommend CI/CD integration:
1. GitHub Actions - GHA workflows + OIDC
2. GitLab CI - GitLab runners on K8s
3. Jenkins - Jenkins on K8s + pipeline templates
4. AWS CodePipeline - Native AWS CI/CD

Consider existing setup, platform, and team workflow.
```

---

16. Cost Optimization Detection

Analysis Algorithm:

```
ANALYZE resource_usage_patterns AND scaling_requirements AND budget_indicators:
  resource_intensity = calculate_resource_needs(
    cpu_intensive_operations,
    memory_usage_patterns,
    io_operations,
    concurrent_users_estimated
  )

  scaling_patterns = analyze_scaling_needs(
    traffic_variability,
    batch_processing,
    real_time_requirements
  )

  return recommend_cost_strategy(resource_intensity, scaling_patterns)
```

Detection Patterns:

- **CPU intensive**: Image processing, cryptography, complex calculations
- **Memory intensive**: Large data processing, caching, in-memory operations
- **I/O intensive**: File processing, database operations, network calls
- **Variable traffic**: Batch jobs, scheduled tasks, event-driven processing

LLM Prompt:

```
Analyze this repository's resource requirements and cost optimization needs:

Resource intensive operations: {cpu_memory_intensive_patterns}
Scaling patterns: {scaling_indicators}
Traffic characteristics: {traffic_patterns}
Performance requirements: {performance_needs}
Budget considerations: {cost_sensitivity_hints}

Recommend cost optimization strategy:
1. Cost-Aware - Resource limits + Spot instances
2. Budget-Constrained - Aggressive scaling + smaller instances
3. Performance-First - Larger instances + dedicated nodes
4. Balanced Approach - Mixed instance types + intelligent scaling

Consider performance vs cost trade-offs.
```

Combined Analysis Workflow

Master Analysis Algorithm:

```
FUNCTION analyze_repository(repo_path):
  # 1. Extract repository data
  file_structure = scan_directory_tree(repo_path)
  package_files = extract_package_files(repo_path)
  code_patterns = analyze_code_patterns(repo_path)
  config_files = extract_config_files(repo_path)

  # 2. Run parallel analysis
  results = run_parallel_analysis([
    analyze_app_type(file_structure, package_files),
    analyze_database_needs(package_files, code_patterns),
    analyze_networking_needs(code_patterns, config_files),
    analyze_storage_needs(code_patterns, file_structure),
    analyze_observability_needs(code_patterns, config_files),
    analyze_ci_cd_needs(file_structure, config_files),
    analyze_cost_optimization(code_patterns, package_files)
  ])

  # 3. Generate recommendations with confidence scores
  return generate_questionnaire_defaults(results)
```

This approach combines **static analysis** with **LLM-powered pattern recognition** to automatically pre-populate the questionnaire with intelligent defaults, while still allowing users to override any decisions.