Spring Boot microservice code analysis strategy Shared by Anand

A Claude code want to learn topics like Code , roles , skills , …

Want to do a code analysis of a spring boot micro service from repo / local workspace

Analysis :
Input : existing package structure and framework used

Multiple version of same endpoint with help of x-version from header

Each may be for different countries some countries have multiple versions

Identify different behaviours, properties

Goal : want to move to single code base, version to support all countries based on config or property

How to do? To define seperate role?

Share the details including, how to prompt.. how to make prompt generic as don't want to feed the structure and framework used again and again for all the micro services

Show less

Claude Code Analysis Guide for Spring Boot Microservices

Overview Strategy

You want to analyze multiple Spring Boot microservices to consolidate versioned endpoints into a config-driven single codebase. Here's how to set up Claude Code effectively:

---

1. Initial Setup & Context Building

First Prompt (One-time per microservice)

bash

```
# Navigate to your microservice directory
cd /path/to/your-microservice

# Initial context-building prompt
claude-code "Analyze this Spring Boot microservice:
1. Map the complete package structure
2. Identify the framework stack (Spring Boot version, dependencies)
3. Find all REST endpoints and their versioning mechanisms
4. Document the project structure in .claude/project_context.md
5. Create a summary of findings in analysis_summary.md"
```

This creates persistent context that Claude Code remembers across sessions.

---

2. Define a Reusable Role/Template

Create a file in your workspace: `.claude/analysis_role.md`

markdown

```
# Microservice Consolidation Analyzer Role

## Your Expertise
You are a Spring Boot microservices consolidation expert specializing in:
- Multi-version API analysis
- Country-specific configuration patterns
- Property-driven architecture design
- Refactoring strategies for version elimination

## Analysis Framework

### Phase 1: Discovery
- Scan for version indicators: @RequestMapping, x-version headers, URL patterns
- Identify country-specific logic patterns
- Map property files and configuration sources
- Document behavioral differences between versions

### Phase 2: Pattern Recognition
- Group similar endpoints by business logic
- Identify configuration candidates (hardcoded values, country codes)
- Find shared vs. unique behaviors
- Analyze dependency variations

### Phase 3: Consolidation Strategy
```

```
- Propose property-based configuration model
- Design country/version resolution strategy
- Suggest feature flag patterns
- Create migration roadmap

## Output Format
Always provide:
1. **Current State Map**: Visual structure of versions/countries
2. **Behavioral Matrix**: What differs between versions
3. **Configuration Model**: Proposed properties schema
4. **Refactoring Plan**: Step-by-step with code examples
5. **Migration Risks**: What to watch for
```

---

3. Generic Prompts (Work for Any Microservice)

Prompt Template 1: Initial Analysis

bash

```
claude-code "Using the analysis_role.md guidelines:
1. Identify all endpoints with versioning (x-version header, URL versioning, or o
2. For each versioned endpoint group, extract:
   - Country-specific behaviors
   - Property/config differences
   - Business logic variations
3. Create a versioning_matrix.csv with columns: endpoint, version, country, key_d
4. Prioritize top 3 endpoints for consolidation pilot"
```

Prompt Template 2: Deep Dive

bash

```
claude-code "Focus on [ENDPOINT_NAME]:
1. Extract all version implementations
2. Diff the logic between versions
3. Identify what should become:
   - Application properties
   - Database configuration
   - Feature flags



4. Generate a consolidated version with config injection
```

```
5. Create test cases covering all original behaviors"
```

Prompt Template 3: Configuration Design

bash

```
claude-code "Design a country-version configuration system:
1. Analyze existing properties files
2. Propose a hierarchical config structure (YAML/properties)
3. Create a CountryVersionConfig class with @ConfigurationProperties
4. Show how to inject this into controllers/services
5. Provide example configs for 2-3 countries"
```

Prompt Template 4: Migration Plan

bash

```
claude-code "Create a migration plan:
1. Sequence the consolidation (which endpoints first)
2. Identify shared utilities to create
3. Propose feature flag strategy for gradual rollout
4. Generate a risk assessment
5. Create a testing checklist"
```

---

4. Making Context Persistent

Create `.claude/project_rules.md`:

markdown

```
# Project Context Rules

## Auto-Load Context
When analyzing this microservice, always reference:
- `project_context.md` for structure
- `analysis_role.md` for methodology
- `versioning_matrix.csv` for current state (if exists)

## Assumptions
- Framework: Spring Boot (detect version from pom.xml/build.gradle)
- Versioning: Look for x-version headers, @RequestMapping paths, package structur
- Config: Check application.yml, application.properties, bootstrap.yml
- Countries: Scan for country codes in: enums, config keys, package names
```

```
## Output Standards
- Always provide runnable code examples
- Use Spring Boot best practices (current stable version)
- Suggest modern patterns: @ConfigurationProperties, profiles, @Conditional
- Include unit test skeletons
```

---

5. Advanced: Multi-Microservice Analysis

If you have multiple microservices to analyze:

bash

```
# Create a workspace-level analysis script
claude-code "Create a script analyze_all_services.sh that:
1. Loops through subdirectories (each microservice)
2. Runs the discovery prompt on each
3. Aggregates findings into a master spreadsheet
4. Identifies common patterns across services
5. Suggests shared libraries for config handling"
```

---

6. Specific Techniques for Your Use Case

Detecting X-Version Headers

bash

```
claude-code "Search codebase for:
1. @RequestHeader('x-version') or similar
2. Custom version interceptors/filters
3. Version resolution logic
4. Document the version routing mechanism"
```

Country-Specific Logic Detection

bash

```
claude-code "Find country-specific code patterns:
1. if/switch statements on country codes
2. Country enum usages

3. Property keys containing country names
```

```
4. Separate service/repository implementations per country
Create a heatmap of where country logic concentrates"
```

Configuration Extraction

bash

```
claude-code "For each identified version difference:
1. Determine if it's a: constant, property, algorithm, external dependency
2. Suggest the appropriate Spring mechanism: @Value, @ConfigurationProperties, St
3. Generate the configuration class structure
4. Show before/after code comparison"
```

7. Example Workflow

bash

```
# Step 1: Initial analysis (run once per service)
cd payment-service
claude-code "Follow .claude/analysis_role.md to analyze this service"

# Step 2: Focus on priority endpoint
claude-code "Consolidate /api/payment/process endpoint versions"

# Step 3: Generate artifacts
claude-code "Create:
1. PaymentConfigProperties.java
2. application-consolidated.yml
3. PaymentController_refactored.java
4. Migration_guide.md"

# Step 4: Validate
claude-code "Generate integration tests that verify all original version behavior
```

8. Key Benefits of This Approach

✅ **No repetitive context**: Role and project rules auto-load
✅ **Generic prompts**: Work across all microservices

✅ **Incremental**: Analyze → Design → Implement → Test per endpoint

✅ **Incremental**: Analyze → Design → Implement → Test per endpoint
✅ **Traceable**: All decisions documented in markdown
✅ **Testable**: Generates tests to prove behavioral equivalence

---

9. Tips for Success

1. **Start Small**: Pick one endpoint with 2-3 versions first
2. **Document Assumptions**: Let Claude Code discover patterns, then validate
3. **Iterate**: Run analysis → review → refine prompt → re-run
4. **Version Control**: Commit `.claude/` directory so context persists
5. **Review Generated Code**: Claude Code excels at patterns but verify business logic

---

10. Sample First Prompt

When starting with a new microservice:

bash

```
claude-code "I'm consolidating versioned endpoints in Spring Boot microservices.

ANALYZE THIS SERVICE:
1. Load .claude/analysis_role.md methodology
2. Discover all versioned endpoints (x-version headers, URL versions)
3. Map country-specific behaviors
4. Create versioning_matrix.csv
5. Recommend top candidate for consolidation pilot

CONTEXT TO REMEMBER:
- Goal: Single codebase, config-driven country/version support
- Look for: hardcoded values, if/switch on country, duplicate endpoints
- Suggest: @ConfigurationProperties, profiles, feature flags

OUTPUT:
- analysis_summary.md with findings
- Next recommended prompt to continue"
```

This approach makes Claude Code your intelligent refactoring partner that learns your codebase incrementally!

A I want to setup in windows using nodejs for the same

Share steps..

Existing application details :

Used hexagon architecture
Package structure / pattern
**/<service name>/<module>_resource - driving layer

Show more

Claude Code Setup for Windows + Node.js

Prerequisites Installation

1. Install Node.js (if not already installed)

```powershell
# Download from https://nodejs.org/ (LTS version recommended)
# Or use winget:
winget install OpenJS.NodeJS.LTS

# Verify installation
node --version
npm --version
```

2. Install Claude Code via NPM

```powershell
npm install -g @anthropic-ai/claude-code

# Verify installation
claude-code --version
```

3. Set Up Anthropic API Key

```powershell
# Set environment variable (session-only)
$env:ANTHROPIC_API_KEY="your-api-key-here"
```

```
# OR set permanently (recommended)
[System.Environment]::SetEnvironmentVariable('ANTHROPIC_API_KEY', 'your-api-key-h

# Verify
echo $env:ANTHROPIC_API_KEY
```

Project Structure Setup

Create Configuration Files

Navigate to your microservice

powershell

```
cd C:\\mymicroservice
```

Create `.claude` directory

powershell

```
mkdir .claude
```

Configuration Files for Hexagonal Architecture

1. `.claude\\analysis_role.md`

markdown

```
# Hexagonal Architecture Microservice Analyzer

## Architecture Understanding
This microservice uses Hexagonal (Ports & Adapters) Architecture:

### Package Structure Pattern
- **Driving Layer**: `**/<service_name>/<module>_resource` (REST controllers, API
- **Core Layer**: `**/<service_name>/<module>` (Domain logic, use cases, ports)
- **Driven Layer**: `**/<service_name>/<module>_adapter` (Database, external serv
- **Configuration**: `**/spring_adapter` (Spring Boot beans, configuration)

### Versioning Pattern

- Version-specific packages: `**/v ? ? ?/<service name>`
```

```
- Version routing: `x-version` header
- Only analyze: `src/main/java` and `src/main/resources`

## Analysis Phases

### Phase 1: Version Discovery
1. Scan `src/main/java` for `v_*` package patterns
2. Identify version-specific resources in each layer:
   - Resource layer: Controllers with version routing
   - Core layer: Version-specific domain logic
   - Adapter layer: Version-specific integrations
3. Map x-version header handling in spring_adapter

### Phase 2: Layer-by-Layer Comparison
For each module, compare across versions:
- **Resource Layer**: Endpoint signatures, request/response DTOs
- **Core Layer**: Business rules, validation differences
- **Adapter Layer**: Database schemas, external API calls
- **Configuration**: Bean definitions, properties per version

### Phase 3: Consolidation Strategy
1. Identify config-extractable differences
2. Propose version resolver strategy in core layer
3. Design country/version configuration properties
4. Create migration plan respecting hexagonal boundaries

## Output Requirements
- Respect hexagonal architecture in refactoring suggestions
- Keep ports/adapters separation
- Suggest configuration in spring_adapter layer
- Maintain testability of core domain logic
```

---

2. `.claude\\project_rules.md`

markdown

```
# Project Analysis Rules

## Scope Restrictions
- **ONLY analyze**: `src/main/java` and `src/main/resources`

- **IGNORE**: `src/test`, `target`, `build`, `.git`, `node_modules`
```

```
## Hexagonal Architecture Constraints
- Changes to core layer must NOT depend on Spring or external frameworks
- Configuration belongs in `spring_adapter` packages
- Version resolution logic should be injectable via ports

## Versioning Detection
- Search pattern: `v_[0-9]_[0-9]_[0-9]` in package names
- Header detection: Look for `@RequestHeader("x-version")` or custom filters
- Country logic: Search for country codes in enums, constants, config keys

## Auto-Load Context
When starting analysis:
1. Load this file's rules
2. Reference `analysis_role.md` for methodology
3. Check for existing `versioning_matrix.csv`
4. Look for `project_context.md` (create if missing)

## Configuration Standards
- Use `@ConfigurationProperties` for version/country configs
- Place config classes in `spring_adapter` layer
- Inject configs into core layer via constructor injection
- Use YAML for hierarchical country/version properties
```

---

3. `.claude\\exclusions.txt`

```
src/test/
target/
build/
.git/
.idea/
*.class
*.jar
node_modules/
.gradle/
```

Single Microservice Analysis

Initial Setup Command

Initial Setup Command

powershell

```
cd C:\\mymicroservice

# Create context file
claude-code "Create a comprehensive analysis:

1. SCAN STRUCTURE:
   - Map all packages under src/main/java
   - Identify hexagonal layers: *_resource, core (**/), *_adapter
   - Find all v_*_*_* version packages
   - List all configuration in spring_adapter

2. VERSION ANALYSIS:
   - Extract all version numbers from package names
   - Find x-version header handling code
   - Document version routing mechanism

3. CREATE ARTIFACTS:
   - Save findings to .claude/project_context.md
   - Create versioning_matrix.csv with columns:
     [version, layer, module, file_path, purpose]
   - Generate initial_assessment.md

Remember: Only analyze src/main/java and src/main/resources"
```

Deep Dive Analysis

powershell

```
claude-code "Analyze version differences:

1. FOR EACH MODULE (group by <module> name):
   - Compare resource layer endpoints across versions
   - Diff core layer business logic
   - Compare adapter layer implementations
   - Extract configuration differences

2. IDENTIFY PATTERNS:
   - What varies by version? (validation rules, DTOs, adapters)

   - What varies by country? (search config, constants, conditionals)
```

```
   - What's identical? (can be shared immediately)

3. OUTPUT:
   - Create module_comparison_<module>.md for each module
   - Update versioning_matrix.csv with 'Difference_Type' column
   - Generate consolidation_candidates.md (ranked by effort vs impact)"
```

Configuration Extraction

powershell

```
claude-code "Design consolidation strategy:

1. PROPOSE CONFIG MODEL:
   - Create VersionCountryProperties.java in spring_adapter
   - Design YAML structure: application-versions.yml
   - Show example for 2 versions, 2 countries

2. REFACTORING PATTERN:
   - How to inject config into core layer (via ports)
   - Version resolver service in core layer
   - Update resource layer to use resolver

3. GENERATE CODE:
   - VersionResolver interface (core layer)
   - VersionResolverImpl (spring_adapter layer)
   - Updated controller example (resource layer)
   - Configuration properties class

Save all to refactoring_proposal/ folder"
```

Multi-Microservice Analysis

Batch Analysis Script

Create: `C:\\multi\\analyze_all.ps1`

powershell

```
# analyze_all.ps1
$microservices = Get-ChildItem -Path "C:\\multi" -Filter "microservice_*" -Direct

foreach ($service in $microservices) {
    Write-Host "Analyzing: $($service.Name)" -ForegroundColor Green

    Set-Location $service.FullName

    # Create .claude directory if missing
    if (-Not (Test-Path ".claude")) {
        New-Item -ItemType Directory -Path ".claude"
    }

    # Copy standard config files
    Copy-Item "C:\\multi\\.claude_templates\\*" -Destination ".claude\\" -Force

    # Run analysis
    claude-code "Quick analysis following .claude/analysis_role.md:
    1. Identify version packages (v_*_*_*)
    2. Count endpoints per version
    3. Find country-specific logic
    4. Output summary to .claude/quick_summary.txt

    Format: ServiceName | Versions | Modules | CountryLogic(Y/N) | Priority(1-5)"

    # Extract summary
    Get-Content ".claude\\quick_summary.txt" | Out-File "C:\\multi\\master_analys
}

Write-Host "`nMaster analysis saved to: C:\\multi\\master_analysis.txt" -Foregrou
```

Setup Template Files

Create: `C:\\multi\\.claude_templates\\` (copy `analysis_role.md` and `project_rules.md` here)

powershell

```
# Create templates directory
mkdir C:\\multi\\.claude_templates

# Copy your configured files
```

```
Copy-Item C:\\mymicroservice\\.claude\\* -Destination C:\\multi\\.claude_template
```

Run Batch Analysis

powershell

```
cd C:\\multi
.\\analyze_all.ps1
```

Aggregate Analysis

powershell

```
cd C:\\multi

claude-code "Analyze master_analysis.txt:
1. Identify common patterns across all microservices
2. Group services by consolidation complexity
3. Suggest shared configuration library structure
4. Create migration_roadmap.md with phases:
   - Phase 1: Pilot (simplest service)
   - Phase 2: Standard services (similar patterns)
   - Phase 3: Complex services (most versions/countries)
5. Estimate effort per service (T-shirt sizes: S/M/L/XL)"
```

Specific Analysis Commands

Find All Version Packages

powershell

```
claude-code "Search src/main/java recursively:
1. Find all directories matching pattern: v_[0-9]_[0-9]_[0-9]
2. For each version directory:
   - List all Java files
   - Identify layer (resource/core/adapter)
   - Extract module name
3. Create version_inventory.csv"
```

Extract Country Logic

powershell

```
claude-code "Find country-specific code:
1. Search for country enums/constants
2. Find if/switch statements on country codes
3. Locate property keys with country names (application.yml)
4. Identify @Profile with country patterns
5. Output country_logic_map.md with file locations and line numbers"
```

Compare Two Versions

powershell

```
claude-code "Compare v_1_0_0 and v_2_0_0 for module 'payment':
1. Resource layer: Diff PaymentResource.java files
2. Core layer: Diff PaymentService.java business logic
3. Adapter layer: Diff PaymentRepository.java queries
4. Configuration: Diff application properties
5. Highlight in comparison_v1_vs_v2_payment.md:
   - Added features (green)
   - Removed features (red)
   - Changed behavior (yellow)"
```

Generate Refactoring Code

powershell

```
claude-code "For module 'customer', generate consolidated version:

INPUT: Analyze existing versions in v_1_0_0, v_1_1_0, v_2_0_0

OUTPUT to refactored/:
1. CustomerResource.java (resource layer)
   - Single controller with version resolution
2. CustomerService.java (core layer)
   - Version-agnostic business logic
3. VersionAwareCustomerAdapter.java (adapter layer)
   - Strategy pattern for version-specific data access

4. CustomerVersionConfig.java (spring adapter)
```

```
   - Configuration properties
5. application-customer-versions.yml
   - Example configs for all versions
6. CustomerServiceTest.java
   - Tests proving all original behaviors preserved"
```

Tips for Windows + Claude Code

1. PowerShell Execution Policy

If scripts don't run:

```powershell
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

2. Long Path Support

Enable long paths (Windows 10/11):

```powershell
# Run as Administrator
New-ItemProperty -Path "HKLM:\\SYSTEM\\CurrentControlSet\\Control\\FileSystem" -N
```

3. Use Windows Terminal

Better experience than CMD:

```powershell
winget install Microsoft.WindowsTerminal
```

4. Git Integration (Optional)

Track Claude's changes:

```powershell
cd C:\\mymicroservice
git init



git add .claude/
```

```
git commit -m "Add Claude Code configuration"
```

Example Workflow

powershell

```
# 1. Navigate to service
cd C:\\mymicroservice

# 2. Initial analysis
claude-code "Follow .claude/analysis_role.md to analyze this hexagonal microservi

# 3. Review output
notepad .claude\\project_context.md

# 4. Deep dive on specific module
claude-code "Analyze the 'order' module: compare all versions, identify consolida

# 5. Generate refactoring
claude-code "Create refactored code for 'order' module in refactored_order/ direc

# 6. Review and test
cd refactored_order
# Review generated files
```

Troubleshooting

Claude Code not found

powershell

```
# Check global npm packages
npm list -g --depth=0

# Reinstall if needed
npm uninstall -g @anthropic-ai/claude-code
npm install -g @anthropic-ai/claude-code
```

API Key issues

powershell

```powershell
# Check if set
echo $env:ANTHROPIC_API_KEY

# Re-set if needed
$env:ANTHROPIC_API_KEY="sk-ant-..."
```

Path too long errors

powershell

```powershell
# Use shorter output directory names
# Enable long paths (see tip #2 above)
```

This setup gives you a repeatable, generic analysis framework for your hexagonal Spring Boot microservices on Windows!

A During analysis of found any new pattern that need to add on .md files ? Improve the prompt / roles / template suggestions also

Enhanced Claude Code Configuration for Hexagonal Microservices

Updated Configuration Files with Pattern Detection

1. `.claude\\analysis_role.md` (Enhanced Version)

markdown

```markdown
# Hexagonal Architecture Microservice Consolidation Expert

## Core Expertise
You are an expert in:
- Hexagonal/Ports & Adapters architecture
- Spring Boot microservices consolidation
- Multi-version API deprecation strategies
- Configuration-driven architecture patterns
- Cross-cutting concern extraction

## Architecture Understanding
```

````
### Package Structure Pattern
````
src/main/java/
├── **/<service_name>/<module>_resource/   # Driving Adapters (REST, Events)
│   └── v_X_Y_Z/   # Version-specific controllers
├── **/<service_name>/<module>/   # Core Domain (Ports, Use Cases)
│   └── v_X_Y_Z/   # Version-specific domain logic
├── **/<service_name>/<module>_adapter/   # Driven Adapters (DB, External APIs
│   └── v_X_Y_Z/   # Version-specific persistence
└── **/spring_adapter/   # Framework Configuration
    └── config/   # Bean definitions, properties
````


### Scope
- **ANALYZE**: Only `src/main/java` and `src/main/resources`
- **IGNORE**: Tests, build artifacts, IDE files

## Analysis Methodology

### Phase 1: Pattern Discovery & Cataloging

#### 1.1 Version Pattern Detection
- [ ] Package-based versions: `v_[0-9]_[0-9]_[0-9]`
- [ ] URL-based versions: `@RequestMapping("/v1/...")`, `@GetMapping("/v2/...")`
- [ ] Header-based versions: `@RequestHeader("x-version")`, `@RequestHeader("X-AP
- [ ] Media type versions: `produces = "application/vnd.company.v1+json"`
- [ ] Custom annotations: `@ApiVersion`, `@VersionMapping`
- [ ] Version interceptors/filters in spring_adapter

**Output**: `version_patterns.md` - Document ALL discovered versioning mechanisms

#### 1.2 Country/Region Pattern Detection
- [ ] Country enums: `CountryCode`, `Region`, `Market`
- [ ] Conditional logic: `if (country == "US")`, `switch(countryCode)`
- [ ] Property prefixes: `payment.us.*`, `config.uk.*`
- [ ] Profile-based: `@Profile("country-de")`, `application-de.yml`
- [ ] Database discriminators: `WHERE country = ?`
- [ ] Strategy patterns: `PaymentStrategy_US`, `ValidationService_UK`
- [ ] Feature flags: `if (featureToggle.isEnabled("feature-x", country))`

**Output**: `country_patterns.md` - Map where and how country logic differs

#### 1.3 Cross-Cutting Concern Patterns
````

- [ ] Validation differences: Custom validators per version/country
- [ ] Error handling: Different error codes/messages
- [ ] Logging: Version-specific log formats
- [ ] Security: Different auth mechanisms per version
- [ ] Audit trails: Version-specific audit requirements
- [ ] Rate limiting: Different limits per version/country
- [ ] Caching strategies: TTL variations
- [ ] Circuit breakers: Timeout differences

Output: `crosscutting_patterns.md`

1.4 Data Model Pattern Detection
- [ ] DTO differences: Field additions/removals between versions
- [ ] Serialization: Custom Jackson modules per version
- [ ] Database schemas: Version-specific tables/columns
- [ ] Migration scripts: Flyway/Liquibase version markers
- [ ] Entity mappings: JPA differences across versions
- [ ] API contracts: OpenAPI/Swagger per version

Output: `data_model_patterns.md`

1.5 Dependency Injection Patterns
- [ ] Version-specific beans: `@Bean("paymentServiceV1")`
- [ ] Conditional beans: `@ConditionalOnProperty("version")`
- [ ] Qualifier usage: `@Qualifier("v2Implementation")`
- [ ] Factory patterns: Version resolution factories
- [ ] Bean post-processors: Version-aware customization

Output: `di_patterns.md`

1.6 Integration Pattern Detection
- [ ] External API versions: Different endpoints per version
- [ ] Message formats: Queue/topic message schemas
- [ ] Async patterns: CompletableFuture vs callbacks
- [ ] Retry mechanisms: Exponential backoff differences
- [ ] Timeout configurations: Per version/country
- [ ] Idempotency handling: Different strategies

Output: `integration_patterns.md`

Phase 2: Impact Analysis

2.1 Behavioral Difference Matrix

````
Create a comprehensive matrix:

| Module | Version | Country | Resource Layer | Core Logic | Adapter Layer | Conf
|--------|---------|---------|----------------|------------|---------------|-----
| payment | v_1_0_0 | US | REST /pay | Legacy validation | Oracle DB | Sync proce
| payment | v_2_0_0 | US | REST /pay | Enhanced fraud check | PostgreSQL | Async
| payment | v_2_0_0 | UK | REST /pay | GDPR consent required | PostgreSQL | Async

**Output**: `behavioral_matrix.csv` and `behavioral_matrix.md`

#### 2.2 Dependency Graph Analysis
- Map version-to-version dependencies
- Identify circular dependencies
- Find orphaned versions (no callers)
- Document version compatibility matrix

**Output**: `dependency_graph.md` with Mermaid diagrams

#### 2.3 Risk Assessment
For each consolidation candidate, assess:
- **Complexity**: Low/Medium/High/Critical
- **Traffic Impact**: % of requests affected
- **Data Risk**: Schema changes required
- **External Dependencies**: Third-party API impacts
- **Testing Effort**: Test case coverage needed
- **Rollback Strategy**: How to revert if needed

**Output**: `risk_assessment.md`

### Phase 3: Consolidation Strategy Design

#### 3.1 Configuration Model Design

**Hierarchical Configuration Structure**:
```yaml
# application-versions.yml
versions:
  defaults:
    timeout: 30s
    retry: 3

  byVersion:
    v1:
````
````

```
      validation: legacy
      processing: synchronous

    v2:
      validation: enhanced
      processing: asynchronous

  byCountry:
    US:
      currency: USD
      taxCalculation: standard
      regulations: [SOX, CCPA]

    UK:
      currency: GBP
      taxCalculation: vat
      regulations: [GDPR, FCA]

  versionCountryOverrides:
    v2:
      UK:
        processing: synchronous  # Override for UK in v2
        additionalValidation: gdprConsent
````

Output: `config_model_proposal.yml` and `ConfigModelDesign.md`

3.2 Architecture Refactoring Pattern

Proposed Structure:
````
src/main/java/
├── <service_name>/<module>_resource/
│   ├── controller/                    # Consolidated controllers
│   │   └── PaymentController.java    # Single controller
│   └── dto/
│       └── versioned/                 # Version-specific DTOs if needed
│
├── <service_name>/<module>/
│   ├── domain/                        # Pure domain models
│   ├── port/
│   │   ├── in/                        # Use case interfaces
│   │   └── out/                       # Repository interfaces
```

```
│   ├── usecase/                    # Business logic implementations
│   └── service/
│       ├── VersionResolver.java    # Version detection
│       └── CountryConfigService.java  # Country-specific config
│
├── <service_name>/<module>_adapter/
│   ├── persistence/
│   │   └── strategy/               # Strategy pattern for version differences
│   └── external/
│       └── versioned/              # Version-specific API clients
│
└── spring_adapter/
    ├── config/
    │   ├── VersionConfiguration.java
    │   └── CountryConfiguration.java
    └── resolver/
        └── VersionCountryResolver.java
````

Output: `refactoring_architecture.md` with examples

3.3 Migration Strategy

Phased Approach:
1. **Phase 0: Foundation** (1-2 weeks)
   - Create configuration infrastructure
   - Build version/country resolver
   - Set up feature flags

2. **Phase 1: Non-Critical Module Pilot** (2-3 weeks)
   - Choose lowest-risk module
   - Implement consolidation pattern
   - A/B test old vs new

3. **Phase 2: Iterative Consolidation** (8-12 weeks)
   - Consolidate modules by priority
   - Deprecate old versions progressively

4. **Phase 3: Cleanup** (2-4 weeks)
   - Remove deprecated code
   - Optimize configuration
   - Update documentation

````
**Output**: `migration_roadmap.md` with Gantt chart

### Phase 4: Code Generation

#### 4.1 Generate Consolidation Artifacts
For each module, create:
- [ ] Consolidated controller with version resolution
- [ ] Version-agnostic domain services
- [ ] Strategy implementations for version-specific behavior
- [ ] Configuration properties classes
- [ ] YAML configuration examples
- [ ] Unit tests with version/country parameterization
- [ ] Integration tests covering all scenarios
- [ ] Migration scripts (if data changes needed)

#### 4.2 Generate Documentation
- [ ] API documentation (OpenAPI 3.0 with version extensions)
- [ ] Configuration guide for DevOps
- [ ] Runbook for troubleshooting
- [ ] Decision log (ADRs - Architecture Decision Records)

## Pattern Recognition Templates

### When You Find a New Pattern:
1. **Document it immediately** in `patterns_discovered.md`
2. **Categorize**: Version-specific | Country-specific | Cross-cutting
3. **Assess prevalence**: How many modules use this pattern?
4. **Propose consolidation**: How to make it config-driven?
5. **Update this role file** with the new pattern for future analyses

### New Pattern Template:
````markdown
## Pattern: [PATTERN_NAME]

**Category**: [Version/Country/CrossCutting/Integration/Data/etc]

**Description**: [What this pattern does]

**Locations Found**:
- Module: [module_name], Files: [file_list]
- Frequency: [X modules, Y files]

**Current Implementation**:
````
````

```java
// Example code
```

Proposed Consolidation:
```java
// Refactored approach
```

Configuration Required:
```yaml
# Config structure
```

Impact Assessment:
- Complexity: [Low/Medium/High]
- Test Coverage Needed: [X test cases]
- Breaking Changes: [Yes/No - describe]
````

## Output Standards

### All Outputs Must Include:
1. **Executive Summary**: 3-5 bullet points
2. **Detailed Findings**: With code examples
3. **Visual Diagrams**: Mermaid for flows, structures
4. **Actionable Recommendations**: Prioritized list
5. **Code Samples**: Runnable, complete examples
6. **Test Strategy**: How to verify behavior equivalence

### Code Quality Standards:
- Follow Spring Boot best practices
- Use constructor injection (not field injection)
- Apply SOLID principles
- Include Javadoc for public APIs
- Use meaningful variable names
- Add TODO comments for manual review points

### Diagram Standards:
Use Mermaid for:
- Architecture diagrams (`graph TD`)
- Sequence flows (`sequenceDiagram`)
- State machines (`stateDiagram-v2`)
````

```
- Entity relationships (`erDiagram`)
- Gantt charts for timelines

## Self-Improvement Protocol

After each analysis:
1. Review `patterns_discovered.md`
2. If 3+ modules share a pattern, add it to this role file
3. Update detection checklist
4. Refine consolidation strategies based on learnings
```

---

2. `.claude\\project_rules.md` (Enhanced)

markdown

```
# Project Analysis Rules & Standards

## Scope Enforcement

### ALWAYS Analyze:
- `src/main/java/**/*.java`
- `src/main/resources/application*.yml`
- `src/main/resources/application*.properties`
- `src/main/resources/db/migration/**` (Flyway/Liquibase)
- `pom.xml` or `build.gradle` (dependency context)

### NEVER Analyze:
- `src/test/**` (unless explicitly requested)
- `target/`, `build/`, `.gradle/`
- `.git/`, `.idea/`, `.vscode/`
- `node_modules/`, `*.class`, `*.jar`
- Log files, temp files

## Hexagonal Architecture Rules

### Layer Boundaries (Strict):
1. **Core Domain** must NOT import:
   - Spring Framework (`org.springframework.*`)
   - Persistence frameworks (`javax.persistence.*`, `org.hibernate.*`)
   - Web frameworks (`javax.servlet.*`)

   - Any infrastructure concerns
```

2. **Resource Layer** may import:
   - Spring Web (`@RestController`, `@RequestMapping`)
   - Core domain (use cases, DTOs)
   - Validation (`javax.validation.*`)

3. **Adapter Layer** may import:
   - Spring Data (`@Repository`, `@Entity`)
   - External client libraries
   - Core domain (port interfaces)

4. **Spring Adapter** orchestrates:
   - All bean definitions
   - Configuration properties
   - Profile management

Dependency Direction:
```
Resource Layer ──> Core Domain <── Adapter Layer
                       ▲
                       │
                Spring Adapter (Configuration)
```

Versioning Detection Protocol

Priority Order:
1. **Header-based**: Check for `x-version`, `X-API-Version`, custom headers
2. **Package-based**: Scan for `v_[0-9]_[0-9]_[0-9]` patterns
3. **URL-based**: Look for `/v1/`, `/v2/` in `@RequestMapping`
4. **Content negotiation**: Check `produces`/`consumes` with version
5. **Custom**: Search for `@ApiVersion`, version resolvers

Country/Region Detection:
1. **Enums**: `Country`, `CountryCode`, `Region`, `Market`
2. **Properties**: Keys like `*.us.*`, `*.uk.*`, `*.de.*`
3. **Profiles**: `@Profile` annotations with country patterns
4. **Conditionals**: String comparisons with country codes
5. **Strategies**: Classes named `*_US`, `*_UK`, etc.

Pattern Discovery Rules

When to Create a New Pattern Document:

```
- Found in **3+ modules** → Create pattern file
- Affects **multiple layers** → Document cross-cutting concern
- **Not in existing patterns** → Add to `patterns_discovered.md`

### Pattern File Naming:
- `pattern_[category]_[name].md`
- Examples:
  - `pattern_version_url_routing.md`
  - `pattern_country_strategy_factory.md`
  - `pattern_crosscutting_audit_trail.md`

## Analysis Output Structure

### Required Directory Structure:
````
.claude/
├── analysis_role.md   # This methodology
├── project_rules.md   # This file
├── project_context.md   # Auto-generated: project overview
├── patterns_discovered.md   # Auto-updated: new patterns found
├── patterns/   # Individual pattern documents
│   ├── pattern_version_*.md
│   ├── pattern_country_*.md
│   └── pattern_crosscutting_*.md
├── analysis/   # Analysis outputs
│   ├── version_patterns.md
│   ├── country_patterns.md
│   ├── behavioral_matrix.csv
│   ├── dependency_graph.md
│   └── risk_assessment.md
├── design/   # Consolidation designs
│   ├── config_model_proposal.yml
│   ├── refactoring_architecture.md
│   └── migration_roadmap.md
└── generated/   # Generated code artifacts
    └── [module_name]/
        ├── consolidated/
        ├── config/
        └── tests/
````

## Auto-Load Context
```

````
On Every Analysis Start:
1. Load `analysis_role.md` methodology
2. Load `project_rules.md` (this file)
3. Check for `project_context.md` (create if missing)
4. Load `patterns_discovered.md` (create if missing)
5. Scan for existing pattern files in `patterns/`

Context Refresh Triggers:
- New pattern found → Update `patterns_discovered.md`
- New module analyzed → Update `project_context.md`
- Configuration changed → Update relevant design docs

Code Generation Standards

Configuration Classes:
````java
@ConfigurationProperties(prefix = "app.version")
@Validated
public class VersionProperties {
    // Use records in Java 17+
    // Include JSR-303 validation
    // Document each property with Javadoc
}
````

Version Resolver:
````java
// Must be in core domain layer
public interface VersionResolver {
    Version resolve(HttpHeaders headers, String path);
}

// Implementation in spring_adapter layer
@Service
public class HeaderBasedVersionResolver implements VersionResolver {
    // Inject configuration, not Spring-specific types
}
````

Strategy Pattern for Variations:
````java
// Core domain
public interface PaymentProcessor {
````
````

```
    PaymentResult process(PaymentRequest request);
}

// Spring adapter - factory
@Configuration
public class PaymentProcessorFactory {
    @Bean
    public Map<Version, PaymentProcessor> processors(...) {
        // Return version-to-implementation map
    }
}
```

Testing Requirements

Every Consolidation Must Include:
1. **Parameterized Tests**: Test all version/country combinations
```java
   @ParameterizedTest
   @CsvSource({
       "v1, US, expected_result_1",
       "v1, UK, expected_result_2",
       "v2, US, expected_result_3"
   })
```

2. **Behavioral Equivalence Tests**: Prove old and new code produce same results

3. **Contract Tests**: Verify API contracts maintained

4. **Performance Tests**: Ensure no regression

Consolidation Safety Checklist

Before generating consolidated code:
- [ ] All version behaviors documented in matrix
- [ ] All country variations identified
- [ ] Configuration model designed and reviewed
- [ ] Test strategy covers all scenarios
- [ ] Rollback plan documented
- [ ] Feature flag strategy defined
- [ ] Migration steps sequenced
- [ ] Team review completed

```
## Documentation Standards

### Every Analysis Must Produce:
1. **Executive Summary** (1 page): Key findings, recommendations
2. **Technical Deep Dive** (5-10 pages): Detailed analysis
3. **Visual Diagrams** (Mermaid): Architecture, flows, dependencies
4. **Code Examples** (compilable): Before/after comparisons
5. **Migration Guide** (step-by-step): How to execute consolidation
6. **Testing Guide**: How to verify correctness

### Markdown Standards:
- Use ATX headers (`#`, `##`, not underlining)
- Code blocks with language specifiers
- Tables for comparisons
- Mermaid for all diagrams
- Links to related documents
- Table of contents for docs > 3 pages

## Self-Learning Protocol

### After Each Microservice Analysis:
1. Review what patterns were unique vs common
2. Update `patterns_discovered.md` with frequency counts
3. If pattern found in 50%+ of services → Promote to `analysis_role.md`
4. Refine detection heuristics based on false positives/negatives
5. Update risk assessment models with actual effort data

### Pattern Promotion Criteria:
- **Frequency**: Found in ≥3 microservices OR ≥50% of analyzed services
- **Impact**: Affects consolidation strategy significantly
- **Complexity**: Non-trivial to detect without explicit guidance
- **Reusability**: Consolidation approach applicable to multiple cases

## Error Handling

### If Analysis Fails:
1. Document the blocking issue in `.claude/blockers.md`
2. Categorize: [Ambiguous | Missing Info | Complex Pattern | Tool Limitation]
3. Suggest workaround or manual steps needed
4. Continue with partial analysis if possible

### If Pattern Unclear:
```

```
1. Document in `.claude/unclear_patterns.md`
2. Provide 2-3 possible interpretations
3. Ask for clarification with specific examples
4. Suggest code locations to manually review
```

---

3. `.claude\\patterns_discovered.md` (New File - Auto-Updating)

markdown

```
# Discovered Patterns Registry

**Last Updated**: [Auto-timestamp]
**Services Analyzed**: [Count]

## Pattern Tracking

| Pattern ID | Pattern Name | Category | Frequency | Services Found | Consolidati
|------------|--------------|----------|-----------|----------------|------------
| P001 | Header-based versioning with x-version | Version | 8/10 | service_1, ser
| P002 | Country enum with switch statements | Country | 6/10 | service_2, servic
| P003 | Version-specific DTO packages | Data Model | 9/10 | service_1, service_2

---

## P001: Header-based Versioning with x-version

**Category**: Version Detection
**First Discovered**: [Service Name] on [Date]
**Frequency**: 8 out of 10 services
**Complexity**: Low

### Description:
Services use custom HTTP header `x-version` to route requests to version-specific

### Implementation Pattern:
````java
@RestController
@RequestMapping("/api/payment")
public class PaymentController {


    @PostMapping("/process")
```
```

```
    public Response process(
        @RequestHeader("x-version") String version,
        @RequestBody PaymentRequest request) {

        if ("1.0".equals(version)) {
            return paymentServiceV1.process(request);
        } else if ("2.0".equals(version)) {
            return paymentServiceV2.process(request);
        }
        throw new UnsupportedVersionException(version);
    }
}
````

Locations:
- `microservice_1`: PaymentResource.java, OrderResource.java
- `microservice_3`: CustomerResource.java
- `microservice_5`: InvoiceResource.java
- ... [list all]

Consolidation Strategy:
```java
// Core domain - version agnostic interface
public interface VersionResolver {
    <T> T resolveImplementation(Class<T> serviceClass, Version version);
}

// Spring adapter - configuration
@Configuration
public class VersionConfiguration {

    @Bean
    public VersionResolver versionResolver(
            Map<Version, PaymentService> paymentServices) {
        return new MapBasedVersionResolver(Map.of(
            PaymentService.class, paymentServices
        ));
    }
}

// Resource layer - clean controller
@RestController
public class PaymentController {
```

```
    private final VersionResolver versionResolver;

    @PostMapping("/process")
    public Response process(
        @RequestHeader("x-version") String versionHeader,
        @RequestBody PaymentRequest request) {

        Version version = Version.parse(versionHeader);
        PaymentService service = versionResolver.resolve(
            PaymentService.class, version
        );
        return service.process(request);
    }
}
```

Configuration Required:
```yaml
versions:
  supported: [1.0, 1.1, 2.0]
  default: 2.0
  deprecation:
    1.0:
      sunset-date: 2025-12-31
      warning-message: "Version 1.0 will be deprecated"
```

Testing Strategy:
- Parameterized test with all supported versions
- Error handling for unsupported versions
- Default version fallback test

Migration Steps:
1. Create VersionResolver interface in core
2. Implement version-to-bean mapping in spring_adapter
3. Refactor controllers to use resolver
4. Add configuration for version management
5. Deploy with feature flag
6. Monitor and validate
7. Remove old if/else version logic

Status: ✅ Standardized (Added to analysis_role.md)

````
---

## P002: Country Enum with Switch Statements

**Category**: Country-Specific Logic
**First Discovered**: [Service Name] on [Date]
**Frequency**: 6 out of 10 services
**Complexity**: Medium

### Description:
Services use Country enum with switch statements scattered throughout business lo

### Implementation Pattern:
```java
public class TaxCalculator {

    public BigDecimal calculate(Order order, Country country) {
        switch (country) {
            case US:
                return order.subtotal().multiply(new BigDecimal("0.07"));
            case UK:
                return order.subtotal().multiply(new BigDecimal("0.20"));
            case DE:
                return order.subtotal().multiply(new BigDecimal("0.19"));
            default:
                throw new UnsupportedCountryException(country);
        }
    }
}
```

### Locations:
- `microservice_2`: TaxCalculator.java (3 locations), ValidationService.java (2 l
- `microservice_4`: PricingEngine.java (5 locations)
- `microservice_7`: ShippingCalculator.java (4 locations)
- ... [list all]

### Problems:
- Violates Open/Closed Principle
- Switch statements duplicated across classes
- Hard to add new countries
- Logic mixed with infrastructure concerns
````

Consolidation Strategy:

```java
// 1. Configuration-driven approach
@ConfigurationProperties("country")
public record CountryProperties(
    Map<String, CountryConfig> configs
) {
    public record CountryConfig(
        BigDecimal taxRate,
        String currency,
        List<String> regulations
    ) {}
}

// 2. Strategy pattern
public interface TaxStrategy {
    BigDecimal calculate(Order order);
}

@Component
public class TaxStrategyFactory {
    private final Map<Country, TaxStrategy> strategies;

    public TaxStrategyFactory(CountryProperties props) {
        this.strategies = props.configs().entrySet().stream()
            .collect(Collectors.toMap(
                e -> Country.valueOf(e.getKey()),
                e -> new ConfigurableTaxStrategy(e.getValue().taxRate())
            ));
    }

    public TaxStrategy getStrategy(Country country) {
        return strategies.get(country);
    }
}

// 3. Usage in domain service
public class TaxCalculator {
    private final TaxStrategyFactory strategyFactory;

    public BigDecimal calculate(Order order, Country country) {
        return strategyFactory.getStrategy(country).calculate(order);
```

```
    }
}
```

Configuration Required:
```yaml
country:
  configs:
    US:
      taxRate: 0.07
      currency: USD
      regulations: [SOX, CCPA]
    UK:
      taxRate: 0.20
      currency: GBP
      regulations: [GDPR, FCA]
    DE:
      taxRate: 0.19
      currency: EUR
      regulations: [GDPR, BaFin]
```

Testing Strategy:
- Parameterized test for each country configuration
- Test strategy selection
- Test unknown country handling
- Property binding validation test

Migration Steps:
1. Extract all switch statements into inventory
2. Create CountryProperties configuration class
3. Implement strategy interfaces
4. Build strategy factory
5. Replace switch statements with strategy calls
6. Add country configs to YAML
7. Test with all countries
8. Remove old switch-based code

Status: 🔄 In Progress (Proven in 2 services)

P003: Version-Specific DTO Packages

Category: Data Model Versioning
First Discovered: [Service Name] on [Date]
Frequency: 9 out of 10 services
Complexity: High

Description:
Each version has separate DTO classes in version-specific packages, even when onl

Implementation Pattern:
````
src/main/java/.../dto/
├── v_1_0_0/
│   ├── PaymentRequest.java   (has: amount, currency, cardNumber)
│   └── PaymentResponse.java  (has: transactionId, status)
├── v_1_1_0/
│   ├── PaymentRequest.java   (has: amount, currency, cardNumber, cvv)
│   └── PaymentResponse.java  (has: transactionId, status)
└── v_2_0_0/
    ├── PaymentRequest.java   (has: amount, currency, tokenId, cvv, metadata)
    └── PaymentResponse.java  (has: transactionId, status, timestamp, fees)
````

Problems:
- 90% code duplication
- Maintenance nightmare (bug fixes need 3x changes)
- Difficult to track what actually differs
- Mapping logic scattered

Consolidation Strategy:

Option A: Single DTO with Optional Fields + Validation Groups
```java
public class PaymentRequest {
    @NotNull(groups = AllVersions.class)
    private BigDecimal amount;

    @NotNull(groups = AllVersions.class)
    private String currency;

    @NotNull(groups = {V1_0.class, V1_1.class})
    @Null(groups = V2_0.class, message = "Use tokenId instead")
    private String cardNumber;
```

```
    @NotNull(groups = {V1_1.class, V2_0.class})
    @Null(groups = V1_0.class)
    private String cvv;

    @NotNull(groups = V2_0.class)
    @Null(groups = {V1_0.class, V1_1.class})
    private String tokenId;

    @Null(groups = {V1_0.class, V1_1.class})
    private Map<String, Object> metadata;

    // Validation groups
    interface V1_0 {}
    interface V1_1 {}
    interface V2_0 {}
    interface AllVersions extends V1_0, V1_1, V2_0 {}
}
```

Option B: Composition with Version-Specific Extensions

```java
// Base DTO - common fields
public class BasePaymentRequest {
    private BigDecimal amount;
    private String currency;
}

// Version-specific extensions
public class PaymentRequestV1 extends BasePaymentRequest {
    private String cardNumber;
}

public class PaymentRequestV1_1 extends PaymentRequestV1 {
    private String cvv;
}

public class PaymentRequestV2 extends BasePaymentRequest {
    private String tokenId;
    private String cvv;
    private Map<String, Object> metadata;
}
```

```
// Mapper registry
@Component
public class PaymentRequestMapper {
    public PaymentDomain toDomain(Object versionedDto, Version version) {
        return switch(version) {
            case V1_0 -> mapFromV1((PaymentRequestV1) versionedDto);
            case V1_1 -> mapFromV1_1((PaymentRequestV1_1) versionedDto);
            case V2_0 -> mapFromV2((PaymentRequestV2) versionedDto);
        };
    }
}
```

Option C: JSON View (Recommended for APIs)

```java
public class PaymentRequest {

    public static class Views {
        public interface V1_0 {}
        public interface V1_1 extends V1_0 {}
        public interface V2_0 {}
    }

    @JsonView({Views.V1_0.class, Views.V1_1.class, Views.V2_0.class})
    private BigDecimal amount;

    @JsonView({Views.V1_0.class, Views.V1_1.class})
    private String cardNumber;

    @JsonView({Views.V1_1.class, Views.V2_0.class})
    private String cvv;
```