PROJECT ROADMAP: WEB-BASED MODERNIZATION PLATFORM

Project: Automated Static Analysis and Visualization Tool
Duration: 8 Weeks (Semester II) | **Framework:** Engineering Design Research (EDR)
Architecture: Local-First Web Platform (FastAPI + Streamlit + Docker)
Team Size: 3 Members

TABLE OF CONTENTS

---

1. EXECUTIVE ARCHITECTURAL VISION

The platform operates as a **local-first application** — all analysis runs on the user's own machine. This ensures that sensitive PHP source code never leaves the user's environment, directly addressing security concerns prevalent in the Tanzanian SME sector.

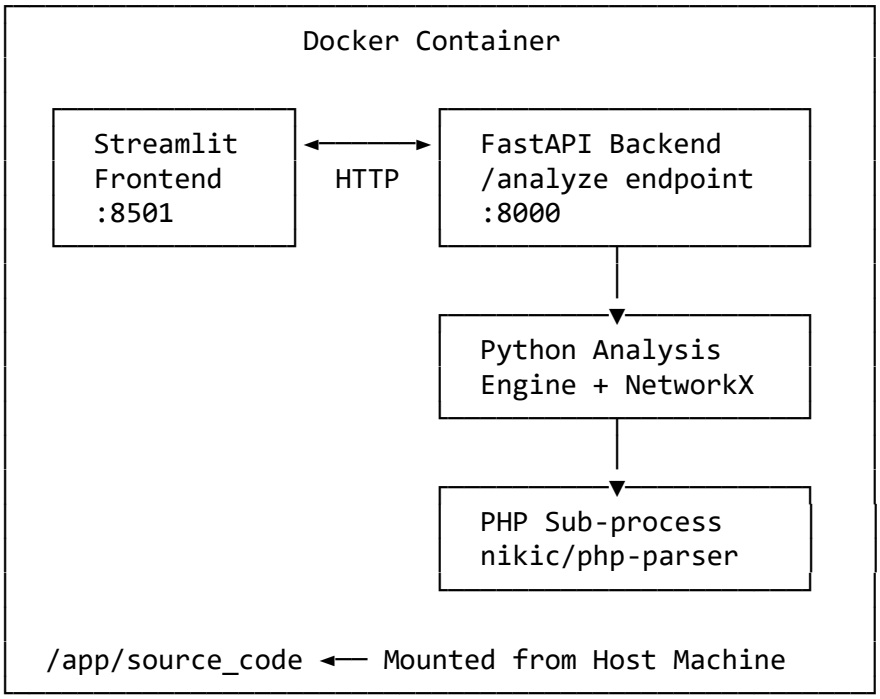
The "Local-First" Deployment Strategy

The system will be packaged as a **single Docker image** containing all dependencies. The user workflow is:

1. Pull and run the Docker container, mounting their legacy project directory to `/app/source_code`
2. Access the interactive dashboard at `http://localhost:8501`
3. Select a directory path, run analysis, and receive visualized insights — all within the browser

The Docker container uses `supervisord` as a process manager to run both the **FastAPI backend** and the **Streamlit frontend** simultaneously. No external internet connection is required after initial setup.

System Architecture Overview



---

2. REFINED TEAM ROLES

Member 1: The Core & Graph Engine

Attribute	Details
Responsibility	Data extraction, AST parsing, and structural graph analysis
Primary Tech	Python, PHP (`nikic/php-parser`), NetworkX
Key Outcome	A robust JSON "Structural Fact" bridge and a mathematical dependency graph with centrality metrics

Core Ownership: - PHP AST Flattener script - NetworkX graph builder and metric calculator - Database access pattern detector - Performance optimization for 1,000+ file codebases

---

Member 2: The Intelligence & Artifact Engine

Attribute	Details
Responsibility	Scoring logic, modernization strategy, and output artifact generation

Attribute	Details
Primary Tech	Python (Data Classes), Jinja2, YAML
Key Outcome	Accurate risk scores, a phased modernization roadmap, and valid OpenAPI/Service stub artifacts

Core Ownership: - Heuristic risk scoring formula and classification tiers - Strangler Fig roadmap generator - Jinja2 templates for OpenAPI YAML and PHP service stubs - Candidate scoring and extraction prioritization system

---

Member 3: The Full-Stack & UX Orchestrator

Attribute	Details
Responsibility	API development, interactive dashboard, Dockerization
Primary Tech	FastAPI, Streamlit (`streamlit-agraph`), Docker
Key Outcome	A seamless, interactive user journey from "Path Selection" to "Artifact Download"

Core Ownership: - FastAPI backend with `/analyze`, `/report`, `/generate` endpoints - Streamlit UI pages and component layout - `streamlit-agraph` dependency graph visualization - Docker multi-stage build and `supervisord` configuration

---

3. THE TECHNICAL BRIDGE (PHP TO PYTHON)

To ensure reliable parsing beyond what regex can provide, the system uses a **Sub-process Bridge** as its primary parsing mechanism.

How It Works

```
Member 1's Python Engine
        │
        │ subprocess.run(["php", "ast_flattener.php", path])
        ▼
PHP Script (nikic/php-parser)
        │
        │ Generates full Abstract Syntax Tree (AST)
        │ Flattens AST → structured JSON
        ▼
JSON Output
{
  "file": "path/to/file.php",
  "classes": [{"name": "UserAuth", "methods": [...], "inherits":
```

```
"BaseAuth"}],
  "functions": [{"name": "validateToken", "calls": ["db_query",
"log_event"]}],
  "imports": ["require_once '../config.php'", "use App\\Models\\User"]
}
        |
        ▼
Python (NetworkX) builds dependency graph from JSON
```

Why Sub-process Bridge Over Regex

Concern	Regex Approach	Sub-process Bridge
Handles heredocs	❌ Fails	✅ Full AST parse
Detects dynamic includes	❌ Misses	✅ AST node traversal
Handles PHP 8+ syntax	❌ Pattern gaps	✅ Parser handles natively
Accuracy on 1000+ file projects	~70%	~95%+

---

4. THE 8-WEEK SPRINT PLAN

PHASE I: Foundation & Web Skeleton (Weeks 1–2)

Goal: Establish the PHP-to-Python data bridge, stand up the FastAPI server, and create the Streamlit landing page skeleton.

WEEK 1

Member 1 (Core & Graph Engine)

Goal: Set up the PHP parsing environment and produce the first working AST extraction

Tasks:

- ☐ Install and configure `nikic/php-parser` via Composer within the project repo
  - Create `/parser/composer.json` with dependency declaration
  - Run `composer install` and verify installation
  - Test parser loads correctly: `php -r "require 'vendor/autoload.php'; echo 'OK';"`
- ☐ Write the initial `ast_flattener.php` script
  - Accept a single PHP file path as a CLI argument
  - Traverse the AST and extract: class names, method signatures, function names, `use` statements
  - Output a flat JSON structure to `stdout`

- ☐ Write a Python wrapper `parser_bridge.py`
  - Call `ast_flattener.php` via `subprocess.run()`
  - Parse the returned JSON into a Python dictionary
  - Handle errors: missing PHP binary, malformed JSON, non-PHP file input
- ☐ Write 3 unit tests for the bridge using small hand-crafted PHP test fixtures

Deliverable: `parser_bridge.py` that can accept a `.php` file path and return a structured Python dict
Integration Point: Share the JSON schema draft with Member 2 for data model alignment

---

Member 2 (Intelligence & Artifact Engine)

Goal: Define all data models and draft the heuristic scoring formula

Tasks:

- ☐ Define Python `dataclasses` for core data models

```python
@dataclass
class Component:
    file_path: str
    name: str
    type: str  # 'class' | 'function' | 'module'
    metrics: dict

@dataclass
class RiskProfile:
    score: float
    level: str  # 'Low' | 'Medium' | 'High'
    factors: list[str]

@dataclass
class Recommendation:
    action: str
    priority: int
    reasoning: str
```

- ☐ Formalize the **Heuristic Scoring Formula**
  - Draft formula: `risk_score = (num_dependencies * 1.0) + (has_db_access * 3.0) + (centrality * 2.0)`
  - Define thresholds: Low (0–3), Medium (3–7), High (7+)
  - Document assumptions and justifications in `SCORING.md`
- ☐ Implement the `RiskClassifier` class
  - Input: a `Component` dict from Member 1's JSON

  - Output: a `RiskProfile` dataclass
  - Stub out DB access flag (default `False` until Week 5)
- ☐ Write unit tests for risk classification
  - Test boundary values at 3 and 7
  - Test with mock component data
  - Verify Low/Medium/High labels are returned correctly

Deliverable: `risk_classifier.py` with unit tests and `SCORING.md` documentation
Integration Point: Share `Component` schema with Member 1 to validate JSON output format alignment

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Stand up the FastAPI server and Streamlit landing page

Tasks:

- ☐ Initialize project structure

```
/project-root
├── /backend          ← FastAPI app
│   ├── main.py
│   └── routes/
├── /frontend         ← Streamlit app
│   └── app.py
├── /parser           ← PHP bridge (Member 1)
├── /analysis         ← Python engine (Member 1 & 2)
├── /templates        ← Jinja2 templates (Member 2)
├── docker-compose.yml
└── requirements.txt
```

- ☐ Set up **FastAPI** backend `main.py`
  - Create `/health` endpoint returning `{"status": "ok"}`
  - Create stub `/analyze` endpoint that accepts `{"path": "/app/source_code"}` and returns mock JSON
  - Enable CORS middleware for Streamlit frontend communication
  - Confirm auto-generated Swagger UI available at `http://localhost:8000/docs`
- ☐ Build the **Streamlit** landing page `app.py`
  - Header with project title and description
  - Directory path input widget (`st.text_input`)
  - "Run Analysis" button (`st.button`)
  - Placeholder output area for results

  - Connect button to POST request to FastAPI `/analyze` stub
- ☐ Install and confirm `streamlit-agraph` renders a 3-node sample graph

Deliverable: Running FastAPI server + Streamlit landing page accessible at `localhost:8501`
Integration Point: Share the `/analyze` request/response schema with Members 1 & 2

WEEK 2

Member 1 (Core & Graph Engine)

Goal: Extend the parser to handle full directories and build the first dependency graph

Tasks:

- ☐ Extend `parser_bridge.py` to recursively scan directories
  - Use `pathlib.Path.rglob("*.php")` to find all PHP files
  - Process each file through `ast_flattener.php`
  - Aggregate all results into a single JSON structure
  - Add a progress counter (file N of total)
- ☐ Enhance `ast_flattener.php` to extract dependency relationships
  - Detect `require_once`, `include`, `require`, `include_once` statements
  - Extract `use` namespace imports
  - Detect function call expressions within methods and map caller → callee
- ☐ Initialize a **NetworkX DiGraph** from the aggregated JSON
  - Nodes: each PHP file/class
  - Edges: each dependency relationship
  - Store raw graph as `graph.json` using `networkx.node_link_data()`
- ☐ Write integration test: parse the `/tests/fixtures/sample_project/` directory and verify the graph has expected nodes and edges

Deliverable: `graph_builder.py` that accepts a directory path and produces a NetworkX graph + `graph.json`
Integration Point: Share `graph.json` schema `{nodes: [...], edges: [...]}` with Member 2 and Member 3

Member 2 (Intelligence & Artifact Engine)

Goal: Integrate graph data into risk scoring and define JSON API contracts

Tasks:

- ☐ Update `RiskClassifier` to accept graph metrics
  - Add `in_degree`, `out_degree`, `betweenness_centrality` fields to `Component`
  - Refine scoring formula weights based on initial test results
- ☐ Define the full **JSON API contract** for the `/analyze` endpoint response

```
{
  "summary": {"total_files": 0, "total_functions": 0, "total_classes":
0},
  "components": [{"file": "", "risk_level": "", "risk_score": 0,
"factors": []}],
  "graph": {"nodes": [], "edges": []},
  "extraction_candidates": []
}
```

- ☐ Write a `report_generator.py` stub
  - Accept the full analysis JSON
  - Output a basic Markdown summary table of components and risk levels
- ☐ Write 5 integration tests using the sample NetworkX graph from Member 1

Deliverable: Updated `risk_classifier.py` accepting graph metrics + `report_generator.py` stub
Integration Point: Share API response schema with Member 3 for Streamlit display planning

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Wire the FastAPI backend to the real analysis engine and display first real results

Tasks:

- ☐ Connect the `/analyze` FastAPI endpoint to Member 1's `parser_bridge.py`
  - Call `parser_bridge.analyze_directory(path)` from the endpoint
  - Return structured JSON matching the agreed API contract
  - Add request validation: verify path exists and contains `.php` files
- ☐ Add error handling to `/analyze`
  - `404` if path not found
  - `400` if no PHP files detected
  - `500` with message if parser subprocess fails
- ☐ Update Streamlit `app.py` to display real analysis results
  - Show summary metrics (`st.metric` cards: total files, functions, classes)
  - Show a raw data table of components with risk levels (`st.dataframe`)
  - Add a loading spinner during analysis (`st.spinner`)

- ☐ Set up Git repository with branch structure
  - `main` (protected), `develop`, `feature/member1-parser`, `feature/member2-scoring`, `feature/member3-ui`

Deliverable: End-to-end flow: Streamlit path input → FastAPI → real parser results displayed in browser
Integration Point: Confirm API contract works with all three members' modules connected

---

PHASE II: Graph Analytics & UI Integration (Weeks 3–4)

Goal: Transform raw dependency data into an interactive visual network with risk overlays.

WEEK 3

Member 1 (Core & Graph Engine)

Goal: Implement full graph metric calculations

Tasks:

- ☐ Implement **In-Degree** calculation
  - In-degree = number of other files that import/depend on this file
  - High in-degree = widely used, high-risk to change
- ☐ Implement **Out-Degree** calculation
  - Out-degree = number of files this file depends on
  - High out-degree = highly coupled, complex to extract
- ☐ Implement **Betweenness Centrality** via NetworkX
  - `nx.betweenness_centrality(G)` — identifies "bridge" files
  - Files with high betweenness are architectural bottlenecks
- ☐ Implement **Cyclic Dependency Detection**
  - Use `nx.simple_cycles(G)` to find all dependency cycles
  - Tag involved nodes with `{"has_cycle": true}`
  - Export cycle list as part of `graph.json`
- ☐ Add **Dead Code Detection** logic
  - Nodes with `in_degree == 0` and not an entry point = orphaned files
  - Export orphan list as `{"orphans": [...]}`

Deliverable: `graph_metrics.py` exporting all metrics per node; updated `graph.json` with metrics
Integration Point: Share updated graph JSON with Member 2 for risk score recalibration

---

Member 2 (Intelligence & Artifact Engine)

Goal: Build the full risk tier categorization system

Tasks:

- ☐ Recalibrate scoring formula with real graph metrics from Member 1
  - Test formula against 20+ sample components
  - Adjust weights if Low/Medium/High distribution is skewed
- ☐ Implement **Dead Code Detection** classification
  - Components with `in_degree == 0` flagged as `"orphan"`
  - Recommendation: `"Safe to delete — no dependents detected"`
- ☐ Categorize all components into risk tiers
  - Add `tier` field: `"extraction_candidate"`, `"refactor_first"`, `"do_not_touch"`
  - `extraction_candidate`: Low risk + Low coupling + No DB access
  - `refactor_first`: Medium risk OR has cycles
  - `do_not_touch`: High risk OR High centrality
- ☐ Generate the first **Strangler Fig Phase Plan** (draft)
  - Phase 1: All `extraction_candidate` components
  - Phase 2: `refactor_first` components after decoupling
  - Phase 3: Core business logic (highest risk)

Deliverable: `tier_classifier.py` with extraction candidate logic + draft phase plan
Integration Point: Share tier data with Member 3 for color-coded graph visualization

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Render the interactive dependency graph in Streamlit

Tasks:

- ☐ Implement `streamlit-agraph` dependency graph
  - Translate `graph.json` nodes/edges into `agraph` Node and Edge objects
  - Color nodes by risk level: 🟢 Low, 🟡 Medium, 🔴 High
  - Size nodes proportionally to their betweenness centrality score
- ☐ Implement **Node Tooltips**
  - On hover: show file name, risk level, risk score, in/out-degree

  - Use `agraph` config to enable tooltip display
- ☐ Add graph controls sidebar
  - Filter by risk level (checkboxes)
  - Filter by file type / directory prefix
  - "Reset View" button
- ☐ Create a second Streamlit page: `pages/2_Graph.py`
  - Graph takes full-width layout
  - Sidebar contains filters

Deliverable: Interactive dependency graph page in Streamlit with color-coded risk nodes and hover tooltips
Integration Point: Confirm node/edge schema with Member 1 for correct graph rendering

---

WEEK 4

Member 1 (Core & Graph Engine)

Goal: Optimize parser performance and handle edge cases

Tasks:

- ☐ Profile parsing performance on a 100-file test project
  - Identify slowest operations
  - Implement parallel file processing using `concurrent.futures.ThreadPoolExecutor`
- ☐ Add file-level caching
  - Cache parsed JSON per file using a hash of its content
  - Skip re-parsing unchanged files on subsequent runs
- ☐ Handle edge cases gracefully
  - Binary files accidentally in source dir (skip with warning)
  - PHP files with syntax errors (log error, continue processing others)
  - Empty PHP files (return empty component, don't crash)
- ☐ Create a `/tests/fixtures/` directory with sample PHP projects
  - `minimal/` — 5 files, no DB, simple dependencies
  - `medium/` — 20 files, 1 DB file, 1 cycle
  - Use these as ground truth for integration tests

Deliverable: Optimized parser that handles 100+ files without crashes, with caching
Integration Point: Share fixture projects with all members for consistent testing

---

Member 2 (Intelligence & Artifact Engine)

Goal: Complete the recommendation engine

Tasks:

- ☐ Build the full **Recommendation Engine**
  - Rule: Low risk + Low coupling → `"Safe to Extract"` + effort estimate
  - Rule: High centrality → `"Refactor Before Extraction"` + specific advice
  - Rule: Cyclic dependency → `"Break Cycle First"` + which files are involved
  - Rule: High in-degree + no DB → `"Extract as Shared Library"`
- ☐ Generate **Effort Estimates**
  - Estimate extraction effort in hours based on: lines of code, number of dependencies, has DB access
  - Formula: `effort_hours = (loc / 50) + (dependencies * 2) + (db_access * 8)`
- ☐ Write integration tests for recommendation engine
  - Test all 4 rule branches
  - Verify effort estimates are reasonable (1–40 hours range)
- ☐ Update `report_generator.py` to include recommendations section

Deliverable: `recommendation_engine.py` with all rules and effort estimates
Integration Point: Feed recommendations into Member 3's "Candidate Detail" view

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Build the Risk Heatmap panel and Candidate Detail view

Tasks:

- ☐ Create **Risk Heatmap** page: `pages/3_Risk_Heatmap.py`
  - Color-coded table: rows = files, columns = risk factors
  - Cell colors: green (safe), yellow (caution), red (danger)
  - Sortable by risk score column
  - Download as CSV button
- ☐ Build **Candidate Detail** view
  - Clicking a node in the graph opens a detail sidebar
  - Shows: file path, risk score breakdown, recommendation, effort estimate
  - Lists all direct dependencies and dependents
- ☐ Add **FastAPI `/report` endpoint**
  - Accepts `{"format": "markdown" | "json"}`
  - Returns analysis report generated by Member 2's `report_generator.py`
- ☐ Wire the "Download Report" button in Streamlit to `/report` endpoint

Deliverable: Risk Heatmap page + Candidate Detail sidebar + working Download Report button

---

MILESTONE 1 (End of Week 4): First Internal Demo

Objective: Demonstrate the full end-to-end analysis flow in the browser

Demo Scenario: 1. Open browser at `localhost:8501` 2. Enter path to `tests/fixtures/medium/` sample project 3. Click "Run Analysis" — loading spinner appears 4. Results page shows: summary metrics, risk-colored dependency graph, risk heatmap 5. Click a node → detail sidebar shows risk breakdown and recommendation 6. Click "Download Report" → Markdown report downloaded

Success Criteria: - ✅ Streamlit UI renders without crashes - ✅ Dependency graph shows correct nodes and edges - ✅ Risk levels correctly classified (Low/Medium/High) - ✅ At least 1 extraction candidate identified - ✅ Report downloads successfully - ✅ All three members can run the full stack on their own machine

Integration Tasks: - Member 3 merges all feature branches into `develop` - All members test on the shared `medium/` fixture project - Document all bugs in GitHub Issues before Week 5

PHASE III: Database Detection & Advanced Logic (Weeks 5–6)

Goal: Add deep insights — DB hotspot detection, the Modernization Roadmap, and advanced extraction logic.

WEEK 5

Member 1 (Core & Graph Engine)

Goal: Detect and catalog all database interaction patterns

Tasks:

- ☐ Detect **Raw SQL queries** via regex on AST string nodes
  - Keywords: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP`
  - Detect in string literals and heredoc syntax
  - Track query location: file path + line number
- ☐ Detect **database connection patterns**
  - `mysqli_connect()`, `new PDO(...)`, `new mysqli()`
  - Track and anonymize any credentials found (replace with `[REDACTED]`)

- ☐ Detect **ORM usage patterns**
  - Doctrine: `EntityManager`, `Repository` class patterns
  - Eloquent/Laravel: `Model::find()`, `->where()`, `->get()` method chains
  - Plain ActiveRecord patterns
- ☐ Tag components with DB access metadata
  - Add to JSON: `"has_db_access": true`, `"db_access_type": ["raw_sql", "orm", "pdo"]`
  - Add `"db_tables_mentioned": ["users", "orders"]` where detectable
- ☐ Create a **Files-to-Tables mapping**
  - Cross-reference which files interact with which DB tables
  - Export as `db_map.json`

Deliverable: Updated `ast_flattener.php` and `parser_bridge.py` with full DB access metadata; `db_map.json`
Integration Point: Share `has_db_access` flags and DB map with Member 2 for risk recalibration

---

Member 2 (Intelligence & Artifact Engine)

Goal: Incorporate DB access into the risk model and build the Modernization Roadmap generator

Tasks:

- ☐ Update risk formula to incorporate DB access
  - Apply DB access multiplier: `has_db_access * 3.0`
  - Differentiate: raw SQL (higher risk) vs ORM (lower risk)
  - Re-run calibration on fixture projects; verify no false positives
- ☐ Build the **DB Migration Planner**
  - For each DB-accessing component: identify tables it touches
  - Suggest API boundary: `"This service should own the 'users' table"`
  - Flag shared table conflicts: multiple components writing the same table = coupling hotspot
- ☐ Build the full **Strangler Fig Roadmap Generator**

```
Phase 1: DB-free utility modules (zero DB access, low coupling)
Phase 2: Services with isolated DB access (owns 1–2 tables only)
Phase 3: Core business logic with shared DB access (requires careful refactoring)
```

  - Generate a Markdown roadmap document with estimated timeline per phase
  - Include risk mitigation strategies per phase

- ☐ Write integration tests for DB-enriched risk scoring

Deliverable: `roadmap_generator.py` producing a 3-phase Strangler Fig Markdown document
Integration Point: Share roadmap JSON structure with Member 3 for the UI Roadmap Panel

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Build the Risk Heatmap panel upgrade and the DB Hotspot view

Tasks:

- ☐ Upgrade the Risk Heatmap with DB access column
  - New column: `DB Access` (🔴 Raw SQL, 🟡 ORM, ⚪ None)
  - Add `DB Tables` column showing tables mentioned per file
- ☐ Create **DB Hotspot Panel**: `pages/4_DB_Analysis.py`
  - Table: files with DB access, query counts, tables accessed
  - Heatmap of which tables are accessed by the most files (coupling hotspot indicator)
  - Warning banner for files with both high centrality AND raw SQL access
- ☐ Add **FastAPI `/db-map` endpoint**
  - Returns `db_map.json` from Member 1's output
  - Used by the DB Analysis page
- ☐ Add **Modernization Roadmap Panel**: `pages/5_Roadmap.py`
  - Display Phase 1 / Phase 2 / Phase 3 cards
  - Each phase card lists extraction candidates with effort estimates
  - Progress bar: X of Y components extracted (mock toggle for now)

Deliverable: DB Analysis page + Modernization Roadmap page in Streamlit
Integration Point: Confirm `db_map.json` schema with Member 1 before building DB Analysis page

---

WEEK 6

Member 1 (Core & Graph Engine)

Goal: Detect modular boundaries and self-contained components

Tasks:

- ☐ Implement **Cohesion Analysis**

  - Detect files with high internal coupling, low external coupling
  - Calculate cohesion score: `internal_calls / (internal_calls + external_calls)`
  - Files with cohesion > 0.7 = strong extraction candidates
- ☐ Detect **Utility Modules**
  - Low dependency count (0–2 total dependencies)
  - No DB access flag
  - Pure functions only (no global state mutations detected)
- ☐ Detect **Bounded Contexts** from directory structure
  - Use directory names as context hints: `/auth/`, `/payment/`, `/notifications/`
  - Cluster related files based on naming conventions and directory
  - Export: `{"context": "auth", "files": [...], "boundary_confidence": 0.85}`
- ☐ Create **Extraction Candidates List**
  - Rank by: Low risk + High cohesion + Low coupling
  - Export as `extraction_candidates.json`: `[{name, reason, priority, effort_hours}]`

Deliverable: `cohesion_analyzer.py` + `extraction_candidates.json`
Integration Point: Share candidates list with Member 2 for validation scoring

---

Member 2 (Intelligence & Artifact Engine)

Goal: Build the candidate validation and prioritization system

Tasks:

- ☐ Create **Candidate Scoring System**
  - Formula: `score = (safety * 0.4) + (business_value * 0.3) + (effort_ease * 0.3)`
  - Safety = inverse of risk score
  - Business value = estimated based on file naming/context (auth, payment = higher value)
  - Effort ease = inverse of effort hours estimate
- ☐ Generate **Top 5 Extraction Plan**
  - Ranked candidate list with full justification per candidate
  - For each: dependencies that must be addressed first, rollback plan note
- ☐ Define **Success Metrics** per candidate
  - Code coverage target before extraction (minimum 60%)
  - Test requirements (at least 1 integration test)
  - Definition of "successfully extracted"
- ☐ Write detailed extraction recommendations

- Step-by-step guide stub for top 3 candidates
- Include interface definition template for each

Deliverable: `candidate_validator.py` with top 5 ranked extraction plan
Integration Point: Feed final candidate list into Member 3's "Candidate Detail" view upgrade

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Implement the "WOW" interactive features — Blast Radius and Modernization Simulator

Tasks:

- ☐ Implement **Blast Radius Tool**
  - In the dependency graph, selecting a node highlights all files affected by its extraction
  - Direct dependents: highlighted in orange
  - Transitive dependents (2nd degree): highlighted in yellow
  - Show count: `"Extracting this module affects X files directly, Y files transitively"`
- ☐ Implement **Orphan Hunter Panel**
  - Dedicated section listing all files with `in_degree == 0`
  - Shows file size and last-modified date
  - "Mark for Deletion" toggle (visual only — does not delete files)
  - Export orphan list as a `.txt` file
- ☐ Add **FastAPI `/extract-candidates` endpoint**
  - Returns `extraction_candidates.json` from Member 1's output with Member 2's scores
- ☐ Upgrade the Candidate Detail sidebar
  - Show Member 2's candidate score breakdown (safety/value/effort)
  - Show step-by-step extraction guide stub
  - Add "Simulate Extraction" button (wires to Modernization Simulator in Week 7)

Deliverable: Blast Radius Tool + Orphan Hunter panel + upgraded Candidate Detail sidebar

---

MILESTONE 2 (End of Week 6): Second Internal Demo

Objective: Demonstrate advanced analysis, DB detection, and the Modernization Roadmap

Demo Scenario: 1. Analyze a real-world open-source PHP project (e.g., a small CMS from GitHub) 2. Show DB Hotspot analysis — identify files with raw SQL and shared table conflicts 3. Display the 3-phase Modernization Roadmap 4. Demonstrate Blast Radius on the highest-centrality node 5. Show the Orphan Hunter panel with files safe for deletion 6. Show Top 5 Extraction Candidates with effort estimates

Success Criteria: - ✅ DB access detection correctly identifies SQL-touching files - ✅ Modernization Roadmap phases are logically ordered - ✅ Blast Radius highlighting works in the graph - ✅ Orphan Hunter correctly lists zero-in-degree files - ✅ Candidate scoring produces a sensible ranked list - ✅ Tool handles real-world codebase (100+ files)

PHASE IV: Artifacts, Polish & Dockerization (Weeks 7–8)

Goal: Generate professional output artifacts, finalize the Docker image, and prepare for final defense.

WEEK 7

Member 1 (Core & Graph Engine)

Goal: Performance profiling, edge case hardening, and validation module

Tasks:

- ☐ Performance profiling: ensure engine can parse **1,000+ files in under 60 seconds**
  - Profile with WordPress core source (~2,000 PHP files)
  - Optimize bottlenecks: use multiprocessing for parallel AST extraction
  - Implement file-level result caching (skip files unchanged since last run)
- ☐ Fix edge cases
  - Malformed PHP syntax: catch parser exception, log warning, skip file
  - Binary files in source directory: detect and skip gracefully
  - PHP files with only comments: handle empty AST output
  - Very large files (10,000+ lines): add timeout per file (default 5s)
- ☐ Create **Validation Module**
  - Verify JSON output schema matches expected contract (use `jsonschema`)
  - Detect incomplete analysis (e.g., 0 functions found in a 500-file project = likely error)
  - Generate a `validation_report.json` summarizing parse success/failure per file

Deliverable: Parser passing performance benchmark; `validation_module.py`
Integration Point: Share `validation_report.json` structure with Member 3 for UI display

---

Member 2 (Intelligence & Artifact Engine)

Goal: Implement Jinja2 artifact generation and finalize the PDF/Markdown report

Tasks:

- ☐ Implement **Jinja2 templates** for OpenAPI YAML generation
  - Template: `openapi.yaml.j2`
  - Auto-populate from extraction candidate: endpoint paths, request/response schemas
  - Include standard GET/POST/PUT/DELETE operations based on detected function names
- ☐ Implement **PHP Service Stub** Jinja2 templates
  - `ServiceClass.php.j2`: basic service class with constructor injection
  - `RepositoryClass.php.j2`: database repository pattern
  - `ControllerClass.php.j2`: REST controller with route annotations
- ☐ Generate the final **Modernization Report** (Markdown + PDF-ready)
  - Executive summary: total files, risk distribution, top 5 recommendations
  - Full component risk table
  - 3-phase roadmap with timelines
  - Appendix: DB access map, orphan file list
- ☐ Write tests for all Jinja2 templates
  - Verify generated PHP is syntactically valid (use PHP lint: `php -l generated_file.php`)
  - Verify generated OpenAPI YAML is valid (use `pyyaml.safe_load()`)

Deliverable: `artifact_generator.py` with all Jinja2 templates; complete Markdown modernization report
Integration Point: Wire Member 3's "Download Artifacts" UI to artifact generator endpoints

---

Member 3 (Full-Stack & UX Orchestrator)

Goal: Build artifact download UI, add Modernization Simulator, polish the full interface

Tasks:

- ☐ Implement **Modernization Simulator**
  - Toggle switch per extraction candidate: "Simulate Extracting This Module"

  - When toggled ON: remove the node and its edges from the graph visualization
  - Update the live metrics: "Removing this module reduces dependency count by X"
  - Show a "Before/After" comparison of graph complexity
- ☐ Add **FastAPI `/generate` endpoint**
  - Accepts: `{"module": "auth", "type": "service" | "openapi" | "all"}`
  - Returns: ZIP file containing generated PHP stubs + OpenAPI YAML
- ☐ Build **Artifact Download Center** in Streamlit
  - Per candidate: buttons for "Download Service Stub (.php)", "Download OpenAPI Spec (.yaml)"
  - "Download Full Report (.md)" button
  - "Download All Artifacts (.zip)" button
- ☐ Final UI Polish
  - Progressive disclosure: collapsed sections expand on demand
  - Responsive layout (sidebar collapses on small screens)
  - Loading states for all async operations
  - Error messages that are human-readable (no stack traces shown to user)
  - Add project logo/branding to header

Deliverable: Modernization Simulator + full Artifact Download Center + polished UI

WEEK 8

All Members (Collaborative)

Goal: Full system integration, final testing, Docker packaging, and defense preparation

Tasks:

Full System Integration

- ☐ Run end-to-end tests on **3 different PHP projects**:
  - `tests/fixtures/minimal/` (5 files, internal)
  - `tests/fixtures/medium/` (20 files, internal)
  - A real-world open-source PHP project (e.g., WordPress 4.0 core or Moodle 2.x)
- ☐ Verify all commands and flows work together without manual intervention
- ☐ Fix all remaining integration bugs from GitHub Issues list
- ☐ Measure precision/recall against manual inspection ground truth (target: 80%+ precision)

Docker Image Build (Member 3 leads)

- ☐ Write `Dockerfile` using multi-stage build

```
# Stage 1: PHP CLI + Composer
FROM php:8.1-cli AS php-stage
RUN composer install ...

# Stage 2: Python + FastAPI + Streamlit
FROM python:3.11-slim
COPY --from=php-stage /app/parser /app/parser
RUN pip install -r requirements.txt
```

- ☐ Write `supervisord.conf` to run FastAPI and Streamlit simultaneously
- ☐ Write `docker-compose.yml` for easy local development
- ☐ Test: `docker build -t modernizer . && docker run -p 8501:8501 -v /my/php/project:/app/source_code modernizer`
- ☐ Verify full analysis pipeline works end-to-end inside the container

Code Quality & Documentation

- ☐ Code cleanup: remove commented-out code, ensure consistent style (run `black` formatter)
- ☐ Add docstrings to all public functions
- ☐ Update `README.md` with: installation guide, Docker usage, screenshots
- ☐ Member 2 writes `EVALUATION.md` documenting precision/recall results

Demo Preparation

- ☐ Prepare 10-minute live demo script
- ☐ Create presentation slides covering: problem, architecture, live demo, results, future work
- ☐ Record backup demo video in case of live demo technical issues
- ☐ Prepare answers to expected supervisor questions

Deliverable: Fully containerized, documented, production-ready MVP

MILESTONE 3 (End of Week 8): Final Demo & Handover

Objective: Present complete MVP to supervisor — demonstrating full analysis pipeline in Docker

Demo Agenda (30 minutes):

1. **Introduction (5 min)** — Problem statement: legacy PHP monoliths in Tanzanian SMEs; solution overview
2. **Live Demo (15 min)**
   - Run Docker container with a real PHP project mounted
   - Show dependency graph with risk coloring
   - Demonstrate Blast Radius on a high-centrality file
   - Show DB Hotspot analysis
   - Display 3-phase Modernization Roadmap
   - Download generated OpenAPI spec and PHP service stub
3. **Technical Deep Dive (5 min)** — Architecture walkthrough: PHP bridge → NetworkX → FastAPI → Streamlit
4. **Results & Metrics (3 min)** — Precision/recall on WordPress 4.0 core; performance benchmark
5. **Future Work (2 min)** — Weeks 9–12 roadmap: JavaScript support, CI/CD integration, cloud deployment

Success Criteria: - ✅ All core features demonstrated without crashes - ✅ Tool correctly analyzes supervisor's test PHP project - ✅ Risk classifications are defensible and accurate - ✅ Generated artifacts (OpenAPI, stubs) are valid and usable - ✅ Docker container runs from a single command - ✅ Supervisor approves 60% completion milestone

---

5. KEY DIVERSITY FEATURES

The following "WOW" features differentiate this project academically and practically:

1. The Blast Radius Tool

When a user selects any file node in the graph, the tool highlights every other file that will be impacted by its extraction. Direct dependents turn orange; transitive dependents turn yellow. A live counter shows total impact scope. This gives developers an immediate understanding of extraction risk beyond a simple score.

2. Orphan Hunter

A dedicated panel listing all files detected as orphans (zero incoming dependencies). Includes file size, path, and a "Mark for Deletion" toggle. Helps teams reduce monolith size *before* the modernization effort begins, reducing scope and risk.

3. Modernization Simulator

An interactive toggle per extraction candidate. When activated, the selected module is hidden from the dependency graph in real time, and metrics update to show how the remaining monolith becomes simpler. This allows architects to *see* the end state before committing to the work.

4. Auto-Generated OpenAPI Specifications

For each extraction candidate, the tool generates a ready-to-use OpenAPI YAML file describing the service's inferred API surface. This saves developers days of manual documentation work and provides a contractual starting point for the new microservice.

---

6. VALIDATION & TESTING STRATEGY

Test Projects

The tool's accuracy will be validated against two well-known open-source PHP codebases: - **WordPress 4.0** — representative of legacy procedural PHP with heavy global state - **Moodle 2.x** — representative of object-oriented PHP with complex class hierarchies

Ground Truth Methodology

1. Manually inspect 50 files from each test project
2. Document actual dependencies, DB access points, and logical modules
3. Compare tool output against manual inspection

Metrics

Metric	Formula	Target
Precision	True Positives / (True Positives + False Positives)	≥ 80%
Recall	True Positives / (True Positives + False Negatives)	≥ 75%
Performance	Analysis time for full WordPress core	< 5 minutes
Stability	Crashes during full WordPress analysis	0 crashes

Testing Layers

- **Unit tests** — each module in isolation (Member 1: parser, Member 2: scorer, Member 3: API)
- **Integration tests** — combined pipeline on fixture projects
- **System tests** — full Docker container run on real-world projects
- **Edge case tests** — empty files, binary files, syntax errors, circular imports

---

7. WEEKLY GAMEPLAN SUMMARY TABLE

Week	Member 1 (Core & Graph)	Member 2 (Intelligence & Artifacts)	Member 3 (Full-Stack & UX)
1	`nikic/php-parser` setup, AST flattener	Define data models, formalize scoring formula, `RiskClassifier`	FastAPI setup, Streamlit landing page, agraph

Week	Member 1 (Core & Graph)	Member 2 (Intelligence & Artifacts)	Member 3 (Full-Stack & UX)
	script, Python bridge		proof-of-concept
2	Directory scanning, NetworkX graph builder, `graph.json` export	Integrate graph metrics into scoring, API contract definition, report stub	Wire FastAPI to real parser, display first real results in Streamlit
3	In/Out-degree, Betweenness Centrality, Cycle detection, Dead code	Risk tier categorization, Strangler Fig draft, orphan classification	Interactive agraph with risk colors, node tooltips, filter sidebar
4	Performance optimization, file caching, edge case handling, fixtures	Recommendation engine, effort estimation, full report generator	Risk Heatmap page, Candidate Detail sidebar, Download Report button
5	SQL detection, DB connection patterns, ORM detection, `db_map.json`	DB risk multiplier, DB Migration Planner, Roadmap generator	DB Hotspot page, Modernization Roadmap page, `/db-map` endpoint
6	Cohesion analysis, utility module detection, bounded contexts, candidates list	Candidate scoring system, top 5 extraction plan, success metrics	Blast Radius tool, Orphan Hunter, upgraded Candidate Detail sidebar
7	1,000-file performance benchmark, edge case hardening, validation module	Jinja2 templates (OpenAPI + PHP stubs), final Markdown report	Modernization Simulator, Artifact Download Center, UI polish
8	End-to-end integration testing, precision/recall measurement	Evaluation report (`EVALUATION.md`), final report review	Docker multi-stage build, `supervisord` config, demo preparation

8. COMMUNICATION & BEST PRACTICES

Version Control

- **Branch strategy:** `main` (protected) → `develop` → `feature/memberN-taskname`
- **Pull Requests:** Every feature merged via PR, requiring at least 1 teammate review
- **Commit convention:** `feat:`, `fix:`, `test:`, `docs:` prefixes for clean history
- **Never commit directly to `main`** — all changes go through `develop` first

API Documentation

- FastAPI auto-generates a **Swagger UI** at `http://localhost:8000/docs`
- All endpoints must have docstrings and typed request/response models
- API contract changes must be communicated in the team sync before implementation

Daily Sync Protocol

- **Format:** Short async standup (Slack/WhatsApp message, not a meeting)
- **Each member posts:** ✅ Done yesterday | 🔨 Working on today | 🚫 Blockers
- **Blocker escalation:** If a blocker isn't resolved within 24 hours, schedule a 15-minute call

Integration Points Calendar

End of Week	Integration Event
Week 1	JSON schema alignment meeting (30 min)
Week 2	First end-to-end pipeline test
Week 4	Milestone 1 internal demo
Week 5	DB access schema alignment
Week 6	Milestone 2 internal demo
Week 7	Full UI + backend wiring session
Week 8	Milestone 3 — Final supervisor demo

This roadmap establishes the complete path to a production-grade, Dockerized modernization tool ready for final defense and potential industry use in Tanzania. Prioritize working software over perfect code — if you fall behind, cut features before cutting quality.

Document Version: 2.0 — Web-Based Platform Edition
Supersedes: FYP_Roadmap_CLI_v1.0.pdf