Pandacea Protocol - Engineering Handbook (v1.1)

Version: 1.1 **Status:** Active

Date: July 12, 2025

Lead Engineer: Asa Conant

1. Purpose

This document defines the standard operating procedures, coding standards, development lifecycle, testing strategy, and security model for the Pandacea Protocol engineering team. Its purpose is to ensure consistency, maintain high quality, foster a collaborative and efficient development environment, and guide the team in implementing proactive security measures. Adherence to this playbook is mandatory for all team members.

2. Core Engineering Principles

- **Secure by Design:** Security is not an afterthought; it is a prerequisite. We will consider the security implications of every feature and architectural decision from the beginning, guided by our Security Threat Model. We will build with the assumption that all external inputs are potentially malicious.
- **Automate Everything:** We automate testing, linting, building, and deployment wherever possible. Automation reduces human error, provides rapid feedback, and frees up developers to focus on solving complex problems. All tests that can be automated, will be.
- **Clarity and Simplicity:** We favor clear, simple, and maintainable code over clever but complex solutions. Code is read far more often than it is written.
- **Pragmatic Execution:** We value progress and delivering value. We will make pragmatic decisions based on the MVP scope, while designing for future scalability.
- **Collective Ownership:** We are all responsible for the quality and success of the entire system. We review each other's code, improve documentation, and help each other solve problems.
- **The Test Pyramid:** We will focus our efforts on building a large base of fast and reliable unit tests, a smaller layer of integration tests, and a very selective set of end-to-end (E2E) tests.
- **Shift-Left Mentality:** Quality is the responsibility of the entire team. Developers are responsible for writing tests for the code they produce.

3. Git Workflow & Ticket Management

We will use the GitFlow branching model to manage our codebase.

- **main branch:** Represents the latest production-ready code. Direct commits are forbidden.
- **develop branch:** The primary development branch for the next release.
- **Feature branches (feature/...):** For new features, branched from develop.
- **Release branches (release/...):** For final testing and release preparation.
- **Hotfix branches (hotfix/...):** For critical production bug fixes, branched from main.

3.1. Commit Messages

We follow the Conventional Commits specification (type(scope): subject).

3.2. Pull Requests (PRs)

All code changes must be submitted via a PR to the develop branch.

- PRs must be linked to a corresponding ticket in our project management tool (GitHub Issues).
- The PR description must clearly explain the "what" and "why" of the change.
- All PRs must be reviewed and approved by at least one other engineer before merging.
- The PR author is responsible for merging after approval.

3.2.1. Code Review Checklist

Reviewers must use the following checklist to ensure consistent and thorough reviews:

- **Correctness:** Does the code meet all requirements of the ticket?
- **Security:** Does the change introduce any vulnerabilities outlined in our Threat Model? Are smart contract interactions safe from reentrancy, etc.?
- **Performance:** Are there obvious performance bottlenecks (e.g., N+1 queries, inefficient loops)?
- **Readability:** Is the code clear, simple, and well-commented where necessary?
- **Test Coverage:** Does the PR include adequate unit and integration tests for the new logic?
- **Documentation:** Have relevant documents (READMES, API Spec, etc.) been updated?

3.3. Ticket Lifecycle

We use GitHub Issues for ticket management. The lifecycle is as follows:

1. **To Do:** A new, prioritized task.
2. **In Progress:** An engineer has assigned the ticket to themselves and created a feature branch.
3. **In Review:** A Pull Request has been opened for the ticket and is awaiting peer review.
4. **Done:** The PR has been approved and merged into develop.

4. Coding Standards & Style Guides

We use industry-standard linters and formatters, enforced via pre-commit hooks and CI.

- **Solidity:** Prettier, Solhint
- **Go:** gofmt, golangci-lint
- **Python:** Black, Flake8
- **JavaScript/TypeScript:** Prettier, ESLint

5. Definition of "Done"

A task is "Done" only when:

1. Code is written and functional.
2. Code passes all linter checks.
3. Unit tests achieve >80% coverage for new/changed code.
4. Integration tests are written for cross-component interactions.
5. All tests pass in the CI pipeline.
6. The PR is approved via the Code Review Checklist.
7. The code is merged into the develop branch.
8. Relevant documentation has been updated.

6. Documentation Standards

- **Code Comments:** Explain the why, not the what.
- **README.md:** Every service must have a README explaining its purpose, setup, and how to run tests.
- **Architectural Documents:** The SDD and API Spec are our source of truth and must be kept up-to-date.

7. Testing Strategy

7.1. Testing Tiers

- **Unit Testing**
  - **Scope:** Testing individual functions or components in isolation.
  - **Coverage Requirement:** >80% for new/modified code.
  - **Tools:** Foundry (Solidity), Go testing, pytest (Python), Jest/React Testing Library (React).
- **Integration Testing**
  - **Scope:** Testing the interaction between two or more components.
  - **Tools:** Docker Compose for environment setup, pytest and Go testing for orchestration.
- **End-to-End (E2E) Testing**
  - **Scope:** Testing a complete user journey from start to finish.
  - **MVP E2E Scenarios:** Successful Lease, Policy Rejection, and Dispute Journeys.
  - **Tools:** Playwright.

7.2. Test Environments

- **Local Development:** Each developer's machine for running unit and integration tests.
- **Continuous Integration (CI):**
  - **Platform:** GitHub Actions.
  - **Triggers:** Runs a full suite of unit and integration tests on every pull request to develop. A PR cannot be merged unless all checks pass.
- **Internal Testnet (Staging):**
  - A persistent environment that mirrors production, deployed on the Polygon Mumbai testnet.
  - This environment is used for E2E tests, performance testing, and manual QA.
  - **Chaos Engineering:** To ensure real-world resilience, our staging environment will incorporate chaos engineering principles. Automated tests will run under adverse conditions including:
    - **Network Fault Injection:** Using tools like pumba to introduce high latency and packet loss between agent containers.
    - **Agent Churn:** Randomly terminating and restarting agent containers during test runs.

7.3. Specialized Testing

- **Smart Contract Testing:**
  - **Fuzz Testing:** Using Foundry's built-in fuzzer to test functions with a wide range of random inputs.
  - **Formal Verification (Post-MVP):** Exploring formal verification tools for critical contracts.
- **Security Testing:**
  - **Threat Model Driven:** Testing will be guided by our Security Threat Model.
  - **Third-Party Audit:** A comprehensive, third-party security audit is mandatory before mainnet launch.
- **Performance & Load Testing:**
  - **Scope:** To validate the non-functional requirements of the system, ensuring scalability and responsiveness under load.
  - **Agent API Load Testing:** We will use tools like k6 or Locust to load test agent API endpoints (e.g., /api/v1/leases) to determine the maximum requests per second a single agent can handle.
  - **System-Level Stress Testing:** We will conduct large-scale simulations on our internal testnet with hundreds of concurrent agents to measure system performance and identify bottlenecks.
- **Economic Model Validation:**
  - **Scope:** To verify the correctness of the protocol's economic mechanisms and test for potential exploits.
  - **Correctness Testing:** Writing specific integration tests to assert correct calculations for the Royalty Distributor contract (RBR) and verify the Dynamic Minimum Pricing (DMP) mechanism.

    - **Exploit Simulation:** Designing and running tests that simulate known economic attack vectors like Reputation Farming and Oracle Manipulation.

7.4. Test Data Management

- **Test Data Generation:** We will maintain a dedicated, version-controlled repository with scripts to generate consistent and realistic test data.
- **Test Environment State Management:** We will implement a "Test Environment Reset" strategy to ensure the reliability of automated tests. Our CI pipeline will include a job that can automatically tear down and re-seed the entire internal testnet, ensuring every test run starts from a pristine, known state.

8. Security Threat Model

8.1. Methodology

We use the STRIDE methodology for technical threats, supplemented with categories for economic, governance, and supply chain threats unique to a decentralized system.

8.2. Technical Threat Analysis (STRIDE)

- **Spoofing (Impersonating someone or something else)**
    - **Threat 1: Agent Impersonation:** A malicious actor spoofs the libp2p Peer ID of a legitimate agent.
        - **Mitigation:** All agent-to-agent communication is authenticated using libp2p's public-key cryptography and all API requests are cryptographically signed.
    - **Threat 2: Smart Contract Impersonation:** A malicious actor deploys a fake protocol contract to defraud users.
        - **Mitigation:** Official contract addresses will be hardcoded in the agent software and SDKs for the MVP, with a future move to a DAO-governed on-chain registry.
    - **Threat 3: Private Key Extraction:** An attacker steals a user's private key.
        - **Mitigation:** The MyData Agent will use hardware-backed key storage where possible (e.g., Secure Enclave, TPM). User education will be a priority.
- **Tampering (Modifying data or code)**
    - **Threat 1: On-Chain State Tampering:** An attacker attempts to alter a user's reputation score or royalty balance.
        - **Mitigation:** Blockchain immutability and strict smart contract access controls prevent unauthorized state changes.
    - **Threat 2: Data Payload Tampering:** A malicious actor modifies a data payload on IPFS.
        - **Mitigation:** IPFS content-addressing (CIDs) ensures data integrity.
- **Repudiation (Claiming you didn't do something)**

    - **Threat 1: Earner Repudiates Consent:** An Earner falsely claims they never gave consent for a data lease.
        - **Mitigation:** Every lease requires a cryptographically signed approval from the Earner, stored on-chain as a non-repudiable record.
    - **Threat 2: Spender Repudiates Payment:** A Spender receives data but refuses to pay.
        - **Mitigation:** The full lease payment is locked in an on-chain escrow contract before any data is exchanged.
- **Information Disclosure (Exposing information)**
    - **Threat 1: Exposing Private Data:** A bug in the agent or PySyft framework exposes raw data.
        - **Mitigation:** Primary defense is the use of Privacy-Preserving Computation (PPC). The MVP scope is limited to federated analysis on extracted features, not raw data, reducing the attack surface.
    - **Threat 2: Network Eavesdropping:** An attacker eavesdrops on P2P communication.
        - **Mitigation:** All agent-to-agent communication will be mandated to occur over an encrypted transport (e.g., TLS 1.3, Noise) supported by libp2p.
- **Denial of Service (DoS) (Denying service)**
    - **Threat 1: DHT Flooding (Sybil Attack):** An attacker creates thousands of fake nodes to flood the DHT.
        - **Mitigation:** Requiring a small PGT stake to publish a DataProduct makes this attack economically costly.
    - **Threat 2: Smart Contract Gas Griefing:** An attacker forces a user to spend large amounts of gas via complex, reverting transactions.
        - **Mitigation:** Adherence to gas-efficient design and the Checks-Effects-Interactions pattern.
- **Elevation of Privilege (Gaining unauthorized capabilities)**
    - **Threat 1: Malicious Agent Gains Control:** An exploit in the MyData Agent's API allows for arbitrary code execution.
        - **Mitigation:** Running all external computation tasks in a sandboxed environment and performing rigorous input validation on all API endpoints.

8.3. Economic & Game-Theoretic Threats

- **Threat 1: Reputation Farming (Collusion):** A group of Sybil agents perform wash-trades of low-quality data to artificially inflate their reputation scores and earn an unfair share of royalties.
    - **Mitigation:**
        1. **Cost of Attack:** Each wash-trade incurs transaction fees, creating a cost floor.
        2. **Algorithmic Detection:** We will develop off-chain monitoring to detect suspicious patterns of activity.
        3. **Reputation Decay:** Reputation scores will have a time-decay component.

- **Threat 2: Oracle Manipulation:** An attacker could manipulate a low-liquidity oracle to influence data prices.
    - **Mitigation:**
        1. **Multiple Oracles:** The PDVF will source data from multiple, reputable oracle providers and use a medianizer.
        2. **Circuit Breakers:** The pricing mechanism will include circuit breakers that pause leasing if prices deviate beyond a predefined percentage.

8.4. Governance Threats

- **Threat 1: Hostile Governance Proposal:** An attacker accumulates a large amount of PGT to pass a malicious proposal.
    - **Affected Components:** DAO, Settlement Layer Contracts.
    - **Mitigation:**
        1. **Time-Lock:** All approved governance proposals will be subject to a mandatory time-lock (e.g., 48-72 hours).
        2. **Quorum Requirement:** A minimum percentage of the total PGT supply will be required to vote on a proposal for it to be valid.
        3. **Community Security Council (CSC) Veto:** The 7-member CSC will have the power to veto a clearly malicious proposal during the time-lock period.

8.5. Software Supply Chain Threats

- **Threat 1: Dependency Hijacking:** An attacker gains control of a third-party package our project uses and publishes a new version with malicious code.
    - **Affected Components:** All software components (Agents, SDKs, etc.).
    - **Mitigation:**
        1. **Locked Dependencies:** We will use lockfiles (go.sum, poetry.lock, package-lock.json) to ensure we always use exact, vetted versions of our dependencies.
        2. **Vulnerability Scanning:** Our CI pipeline will automatically scan all dependencies for known vulnerabilities using tools like Snyk or GitHub's Dependabot.
- **Threat 2: Build Process Compromise:** An attacker compromises our CI/CD pipeline and alters the final build artifacts.
    - **Mitigation:**
        1. **Strict Access Controls:** Access to the production deployment environment and its secrets will be restricted to a minimal number of authorized personnel.
        2. **Signed Commits:** All developers will be required to sign their commits.

9. Developer Onboarding

9.1. Environment Setup Checklist

- [ ] Install core languages: Go, Python (via Poetry), Node.js.
- [ ] Install core tools: Docker, Git, VS Code with recommended extensions.
- [ ] Install blockchain tools: Foundry.
- [ ] Clone all required project repositories.
- [ ] Configure environment variables using the template .env.example and secrets from Doppler.
- [ ] Run
  `make setup` in the root of each repository to install dependencies.

9.2. First Day Tasks

1. Successfully run the entire test suite for all services locally.
2. Be added to the GitHub organization and relevant repositories.
3. Receive an invitation to our secrets management platform (Doppler).
4. Pick a small, well-defined "good first issue" to complete the full ticket lifecycle.

10. Dependency Management

- We use go.mod for Go, poetry for Python, and npm for Node.js.
- Dependencies should be updated deliberately, not automatically. A dedicated "hardening" period will be scheduled before each major release to update and test all dependencies.
- Adding a new dependency requires approval from the Lead Engineer to assess its security and maintenance implications.

11. Secrets Management

- **Rule Zero:** No secrets are ever to be committed to the Git repository. This includes API keys, private keys, passwords, etc..
- **Development:** We use Doppler to manage secrets for local development. Each developer will have read-only access to the development environment secrets.
- **CI/CD & Production:** We use GitHub Encrypted Secrets for our CI/CD pipelines and production environments. Only the Lead Engineer will have permission to manage production secrets.

12. Related Documents

- System Design Document (v1.1)
- API Spec (v1.1)
- Engineering Playbook (v1.1)