**AI Marketplace Project: Guidelines & Standard Operating Procedures (SOPs)**

**Version:** 1.0 **Date:** April 23, 2025 **Review Cycle:** Quarterly or as needed

**1. Introduction & Purpose**

* **Purpose:** To ensure consistency, maintain high code quality, streamline the development process, facilitate effective collaboration, and simplify onboarding for new team members working on the AI Marketplace project.
* **Scope:** These guidelines cover the entire development lifecycle, from planning and coding to testing and deployment.
* **Guiding Principles:**
  + **User-Centricity:** Prioritize a clean, intuitive, and responsive user experience.
  + **Minimalism:** Strive for simplicity and clarity in design and code.
  + **Quality:** Write clean, maintainable, well-tested code.
  + **Collaboration:** Foster open communication and mutual support.
  + **Agility:** Adapt to changing requirements while maintaining structure.

**2. Development Workflow & Sprint Planning**

* **Methodology:** We will follow a modified Agile approach, using sprints (e.g., 1-2 weeks) to manage work.
* **Task Management:** Use a shared tool (e.g., Trello, Asana, Jira, GitHub Projects) to track tasks (features, bugs, chores). Each task should have a clear description, acceptance criteria, and an assigned owner.
* **Sprint Cycle:**
  1. **Sprint Planning:** Define sprint goals and select tasks from the backlog. Estimate effort (optional, can use story points or time).
  2. **Development:** Team members work on assigned tasks, following branching and coding standards (see below).
  3. **Daily Stand-ups:** Quick (5-10 min) daily sync: What did you do yesterday? What will you do today? Any blockers?
  4. **Testing:** Continuous testing throughout the sprint (Unit, Integration).
  5. **Code Review:** All code must be reviewed before merging (see Section 5).
  6. **Sprint Review/Demo:** Showcase completed work at the end of the sprint.
  7. **Sprint Retrospective:** Discuss what went well, what didn't, and how to improve in the next sprint.
  8. **UAT Phase:** After a sprint (or set of sprints), conduct User Acceptance Testing on a staging environment before release.

**3. Coding Standards**

* **Languages:** HTML, CSS, JavaScript (Specify frameworks/libraries, e.g., React, Vue, Node.js, etc., if applicable).
* **Formatting:**
  + Use an automated formatter (e.g., Prettier) integrated into your IDE/workflow to ensure consistent style.
  + Configure linters (e.g., ESLint for JS, Stylelint for CSS) to catch potential errors and enforce style rules.
  + *Key Rules (Examples - Define your specifics):*
    - Indentation: 2 spaces (or 4, be consistent).
    - Quotes: Single quotes (') for JS, double quotes (") for HTML attributes.
    - Line Length: Max 100-120 characters.
    - Semicolons: Required at the end of JS statements.
* **Naming Conventions:**
  + camelCase for JavaScript variables and functions (e.g., getUserData).
  + PascalCase for JavaScript classes and components (e.g., UserProfileCard).
  + kebab-case for CSS classes and HTML IDs/attributes (e.g., user-profile-card, data-user-id).
  + UPPER\_SNAKE\_CASE for constants (e.g., MAX\_RESULTS).
  + Use descriptive names for variables, functions, classes, and files.
* **Comments:**
  + Comment complex logic, assumptions, or workarounds. Explain *why* something is done, not just *what*.
  + Use // TODO: for planned work and // FIXME: for known issues needing fixes.
  + Avoid commented-out code; use version control history instead.
* **HTML:**
  + Use semantic HTML5 elements (<header>, <nav>, <main>, <article>, <aside>, <footer>, etc.).
  + Ensure accessibility (use alt tags for images, proper labels for forms, ARIA attributes where necessary).
* **CSS:**
  + Use a consistent methodology (e.g., BEM, SMACSS, or CSS Modules).
  + Prefer classes over IDs for styling.
  + Keep selectors specific but avoid excessive nesting.
  + Organize CSS logically (e.g., by component).
* **JavaScript:**
  + Follow ES6+ best practices (e.g., use let/const over var, arrow functions, modules).
  + Write small, focused functions/modules (Single Responsibility Principle).
  + Avoid global variables where possible.
  + Handle errors gracefully.
* **DRY (Don't Repeat Yourself):** Abstract reusable code into functions, components, or utilities.
* **Performance:** Optimize assets (images, fonts), minimize HTTP requests, consider lazy loading.

**4. Version Control (Git)**

* **Repository:** Use a central repository (e.g., GitHub, GitLab, Bitbucket).
* **Branching Strategy:**
  + main: Represents production-ready code. Only merge tested, reviewed code here (usually from develop).
  + develop: Integration branch. Features are merged here after review. This branch should always be stable and potentially deployable to staging.
  + feature/<feature-name>: Create a new branch from develop for each new feature or significant change (e.g., feature/user-dashboard, feature/tool-search-bar). Use descriptive names.
  + fix/<issue-description>: For bug fixes branched off develop (or main for hotfixes).
* **Commits:**
  + Commit small, logical units of work frequently.
  + Write clear, concise commit messages using a standard format (e.g., Conventional Commits: feat: Add user login functionality, fix: Correct alignment on tool cards, refactor: Improve search algorithm, docs: Update README).
  + The first line should be a short summary (<= 50 chars). Add more detail in the body if needed.
* **Pull Requests (PRs) / Merge Requests (MRs):**
  + All code changes intended for develop (or main) *must* go through a PR.
  + Push your feature or fix branch to the remote repository and create a PR targeting develop.
  + Clearly describe the changes in the PR, including *what* it does, *why* it's needed, and *how* to test it. Link to the relevant task/issue if applicable.
  + Assign at least one team member for review.
* **Code Reviews:**
  + Reviewers should check for adherence to coding standards, correctness, potential bugs, performance issues, security concerns, and maintainability.
  + Provide constructive feedback. Be specific and suggest improvements.
  + The author should address feedback before the PR is approved.
  + Once approved, the author (or reviewer, based on team agreement) merges the PR into develop. Prefer "Squash and Merge" or "Rebase and Merge" to keep the develop history clean, but agree on a strategy.
* **Keeping Branches Updated:** Regularly pull changes from develop into your feature branches (git pull origin develop or git rebase origin/develop) to avoid large merge conflicts later.

**5. Testing Strategy**

* **Goal:** Ensure application correctness, stability, and usability across different scenarios and devices.
* **Unit Testing:**
  + **Who:** Developers write tests alongside their code.
  + **What:** Test individual functions, methods, or components in isolation. Mock dependencies.
  + **Tools:** (e.g., Jest, Vitest, Mocha for JS; PyTest for Python).
  + **When:** Run automatically on commit/push (if CI/CD is set up) and before creating PRs. Aim for meaningful coverage of critical logic.
* **Integration Testing:**
  + **Who:** Developers or dedicated QA.
  + **What:** Test the interaction between different units/modules (e.g., component interaction, component-API interaction).
  + **Tools:** Can use unit testing frameworks or dedicated tools (e.g., React Testing Library for component integration, Supertest for API endpoints).
  + **When:** Run after unit tests pass, often as part of the CI pipeline.
* **End-to-End (E2E) Testing:**
  + **Who:** Developers or dedicated QA.
  + **What:** Simulate real user scenarios by interacting with the application through the UI. Covers critical user flows (e.g., user registration, searching for a tool, adding to favorites).
  + **Tools:** (e.g., Cypress, Playwright, Selenium).
  + **When:** Run less frequently (e.g., before deployments to staging/production) as they are slower and more brittle.
* **User Acceptance Testing (UAT):**
  + **Who:** Project stakeholders, end-users, or designated internal testers (can be you and your friend initially).
  + **What:** Verify that the implemented features meet the business requirements and user expectations from an end-user perspective. Use pre-defined test scenarios based on requirements.
  + **When:** Conducted on a staging environment before a production release. Feedback/bugs are reported back to the development team.
* **Manual Testing:**
  + **Responsiveness:** Manually test layouts on various screen sizes (use browser dev tools and real devices if possible). Target key breakpoints (mobile, tablet, desktop).
  + **Browser Compatibility:** Test on agreed-upon target browsers (e.g., latest Chrome, Firefox, Safari, Edge).
  + **Exploratory Testing:** Manually explore the application to find edge cases or usability issues not covered by automated or scripted tests.
* **Bug Tracking:** Use the chosen task management tool or GitHub Issues to log bugs with clear steps to reproduce, expected vs. actual results, environment details (browser, OS), and screenshots/videos if helpful. Assign severity/priority.

**6. Deployment**

* **(Define your process)** Even if initially manual:
  + Ensure all tests pass on the develop branch.
  + Merge develop into main.
  + Tag the release in Git (e.g., v1.0.0).
  + Deploy the main branch code to the production server.
  + Perform post-deployment checks (smoke tests) to ensure the live site is working.
* **Environments:** Aim for at least:
  + **Local:** Developer machines.
  + **Staging (Optional but Recommended):** A production-like environment for UAT and final testing.
  + **Production:** The live site.

**7. Communication & Collaboration**

* **Primary Channel:** (e.g., Slack, Microsoft Teams, Discord). Use dedicated channels (e.g., #general, #development, #bugs).
* **Meetings:** Keep meetings focused (Daily Stand-ups, Sprint Planning, Review, Retrospective).
* **Knowledge Sharing:** Document important decisions, architecture choices, and setup instructions (e.g., in the project README or a shared wiki).
* **Respectful Feedback:** Be open to giving and receiving constructive criticism during code reviews and discussions. Assume positive intent.
* **Availability:** Communicate your working hours and availability, especially if working remotely or flexibly.

**8. Documentation**

* **README.md:** Essential. Include project description, setup instructions (dependencies, environment variables), how to run the project locally, how to run tests, and key architectural decisions.
* **Code Comments:** As per Section 3.
* **API Documentation (if applicable):** Document any APIs your frontend consumes or your backend provides (e.g., using Swagger/OpenAPI).

**9. Review & Updates to this SOP**

* This document is a living guide.
* Review and update these guidelines quarterly or whenever significant changes occur in the project, team structure, or technology stack.
* Propose changes via discussion or a PR to this document if it's version controlled.

**Agreement:**

All team members should read, understand, and agree to follow these guidelines.

* Varun Kumar
* Sachin Yadav