Engineering Guidelines & Coding Standards

Domain: Core Engineering
Scope: All Technical Staff

1. PHILOSOPHY
At Intra, we build systems that think. Our codebases must be resilient, deterministic where possible, and rigorously tested. Unlike traditional software, Agentic AI applications introduce non-deterministic outputs. Our engineering standards are designed to mitigate these risks through strict control flows and observablity.

2. GENERAL CODING STANDARDS
2.1 Languages
Primary languages: Python (Backend/AI), TypeScript (Frontend/Edge).
Legacy C++ code is being deprecated; do not start new modules in C++.

2.2 Style Guides
- Python: Follow PEP 8 strictly. Use 'Black' for formatting and 'Ruff' for linting.
- TypeScript: Follow the Google TypeScript Style Guide. Use Prettier.

2.3 Typing
Strong typing is mandatory. Python code must utilize type hints (via the `typing` module) and pass `mypy` checks in strict mode. `Any` should be avoided unless absolutely necessary.

3. AI-SPECIFIC ARCHITECTURE

3.1 Agent Design
Agents should be stateless. Context must be retrieved from the Vector Store or Redis cache at runtime. Do not store conversation history in local memory variables, as this prevents horizontal scaling.

3.2 Prompt Engineering & Versioning
Prompts are code. They must be stored in the `prompts/` directory as YAML or JSON files. Hardcoding prompts inside Python functions is prohibited. We use semantic versioning for prompts (e.g., `system_prompt_v1.2.yaml`).

3.3 Hallucination Guardrails
All LLM outputs must be validated against a schema (using Pydantic or Zod) before being processed by downstream services. If validation fails, the agent must trigger a retry mechanism up to 3 times before raising a `ModelOutputError`.

4. VERSION CONTROL

4.1 Branching Strategy
We follow a modified Git Flow.
- `main`: Production-ready code.
- `develop`: Integration branch.
- `feature/name`: New features.
- `fix/name`: Bug fixes.

4.2 Commit Messages
Use Conventional Commits (e.g., `feat: add memory module`, `fix: resolve token limit error`). Messages must be imperative and clearly describe the change.

4.3 Pull Requests
PRs require at least two approvals. One approval must come from a Senior Engineer. All CI checks (Lint, Test, Security Scan) must pass before merging.

5. TESTING STRATEGY

5.1 Unit Testing
Target 85% code coverage. Use `pytest` for Python. Mock all external API calls (OpenAI, AWS, Pinecone) using `unittest.mock`.

5.2 Evaluation (Evals)
For AI components, unit tests are insufficient. You must write 'Evals' that run the agent against a dataset of known inputs and expected outcomes. Measure metrics such as adherence to instructions, toxicity, and latency. These run nightly.

6. DOCUMENTATION

6.1 Code Documentation
Public functions must have docstrings in Google format. Include arguments, return types, and exceptions raised.

6.2 API Documentation
REST APIs must be documented using OpenAPI (Swagger). GraphQL endpoints must have schema descriptions.

7. DEPLOYMENT & OBSERVABILITY
7.1 CI/CD Pipelines
Deployments are automated via GitHub Actions. Do not manually SSH into servers to deploy code.

7.2 Logging
Use structured JSON logging. Logs must include `trace_id` to correlate requests across microservices. Sensitive data (PII, API Keys) must be masked in logs.