I. Language-Specific Conventions:

- **Python:** Adhere to PEP 8 (https://www.python.org/dev/peps/pep-0008/) with the following additions/modifications:
  - Maximum line length: 120 characters (for improved readability on wider screens).
  - Use type hints (https://docs.python.org/3/library/typing.html) extensively for function arguments, return values, and variable declarations. This is crucial for understanding data flow in API orchestrations.
  - Docstrings should follow the Google Style Guide (https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings) for consistency and automated documentation generation.
- **JavaScript (if applicable for frontend/UI components):** Adhere to a widely-used style guide like the Airbnb JavaScript Style Guide (https://github.com/airbnb/javascript) with consistent use of semicolons and preferably using a linter like ESLint.

II. Naming Conventions:

- **Variables:**
  - Use descriptive names in snake_case (e.g., api_response, user_data, request_payload).
  - Avoid single-letter variable names (except for very short loop counters).
  - For AI/ML related variables, include prefixes like model_, feature_, or prediction_ (e.g., model_weights, feature_vector).
- **Functions/Methods:**
  - Use descriptive names in snake_case (e.g., process_api_data, validate_user_input, predict_api_latency).
  - For functions related to API interactions, include the API name or a clear abbreviation (e.g., get_user_from_auth_api, parse_google_maps_response).
- **Classes:**
  - Use PascalCase (e.g., APIOrchestrator, DataManager, PredictionModel).
  - Classes related to specific APIs should include the API name (e.g., GoogleMapsAPIClient).
- **Files:**
  - Use lowercase with underscores or hyphens (e.g., api_orchestration.py, data_processing_utils.py).
  - For modules related to specific APIs, include the API name (e.g., google_maps_integration.py).
- **Constants:**
  - Use uppercase with underscores (e.g., MAX_RETRIES, API_TIMEOUT).

III. Code Structure and Organization:

- **Modular Design:** Break down complex orchestrations into smaller, reusable functions and classes.
- **API Interactions:** Encapsulate API calls within dedicated functions or classes to promote code reusability and maintainability. Handle API authentication, error handling, and data transformation within these modules.
- **AI/ML Components:** Separate AI/ML model training, prediction, and evaluation logic into dedicated modules. Use established ML frameworks (e.g., TensorFlow, PyTorch) and follow their best practices.
- **Error Handling:** Implement robust error handling using try-except blocks. Log errors with detailed information (including timestamps, API request details, and relevant context). Avoid generic except clauses; catch specific exceptions whenever possible.
- **Logging:** Use a logging library (e.g., Python's logging module) for consistent logging throughout the application. Log important events, errors, and performance metrics.

- **Configuration**: Store configuration settings (API keys, endpoints, model paths) in a separate configuration file (e.g., .ini, .yaml, or environment variables) to facilitate easy modification without code changes.

IV. Documentation:

- **Docstrings**: Write comprehensive docstrings for all functions, classes, and modules, explaining their purpose, arguments, return values, and any exceptions they might raise. Use the Google Style Guide for docstrings.
- **Comments**: Use comments sparingly to explain complex logic or non-obvious code. Avoid redundant comments that simply restate the code.
- **README**: Maintain a clear and up-to-date README file for the project, explaining its purpose, architecture, dependencies, and how to run it.
- **API Documentation**: Generate API documentation automatically from the code (e.g., using Sphinx or similar tools).

V. Version Control:

- Use Git for version control.
- Follow a clear branching strategy (e.g., Gitflow).
- Write meaningful commit messages that describe the changes made.

VI. Testing:

- Write unit tests for all core functionalities, including API interactions, AI/ML components, and orchestration logic. Use a testing framework like pytest (Python) or Jest (JavaScript).
- Aim for high test coverage.
- Implement integration tests to verify the interaction between different components

VII. Code Reviews:

- Conduct thorough code reviews before merging any changes into the main branch.
- Focus on code quality, adherence to conventions, and potential issues.

VIII. TEDDi Specific Conventions:

- **API Orchestration Flows**: Clearly document the orchestration logic, including the sequence of API calls, data transformations, and error handling strategies. Consider using visual diagrams or flowcharts to illustrate complex orchestrations.
- **AI Model Integration**: Document the AI models used, their training data, performance metrics, and how they are integrated into the orchestration process.
- **Data Transformation**: Clearly define the data transformation steps between API calls. Document the format of the input and output data for each transformation.
- **Performance Optimization**: Document any performance optimization techniques used, such as caching, asynchronous calls, or load balancing.

IX. Enforcement:

- Use linters and formatters (e.g., flake8, mypy, autopep8 for Python; ESLint, Prettier for JavaScript) to automatically enforce coding style.
- Integrate these tools into the development workflow (e.g., as part of the CI/CD pipeline).