/core\_services/

├── email/

│ ├── parser/

│ │ ├── subject\_parser.py # Extracts and processes email subjects.

│ │ ├── body\_parser.py # Extracts and processes email body content.

│ │ ├── metadata\_extractor.py # Extracts sender, recipient, and timestamp metadata.

│ │ └── attachment\_handler.py # Processes email attachments securely.

│ ├── sender.py # Manages SMTP connections for sending emails.

│ └── receiver.py # Handles IMAP connections for fetching emails.

├── workflow/

│ ├── rules/

│ │ ├── rule\_loader.py # Loads workflow rules from YAML files.

│ │ ├── rule\_validator.py # Validates rule configurations for correctness.

│ │ └── rule\_executor.py # Executes rules on tasks based on matching conditions.

│ ├── escalations/

│ │ ├── escalation\_rules.py # Defines escalation conditions and thresholds.

│ │ └── escalation\_handler.py # Handles task escalations and notifications.

│ └── workflow\_manager.py # Orchestrates workflows and task routing.

├── database/

│ ├── connectors/

│ │ ├── postgres\_connector.py # Manages PostgreSQL connections.

│ │ └── sqlite\_connector.py # Manages SQLite connections for offline operations.

│ ├── schema\_initializer.py # Initializes database schema.

│ └── operations.py # Performs CRUD operations and queries.

├── logging/

│ ├── activity\_logger.py # Logs workflow and task actions.

│ ├── error\_logger.py # Logs system errors for debugging.

│ └── security\_logger.py # Logs security-related events.

└── task/

├── scheduler.py # Handles scheduling of tasks.

├── executor.py # Executes tasks and updates statuses.

└── notifier.py # Sends notifications related to tasks.

/config/

├── organizations/

│ ├── default\_config.yaml # Default settings for workflows and rules.

│ ├── school\_config.yaml # Configuration tailored for schools.

│ ├── hospital\_config.yaml # Configuration tailored for hospitals.

│ ├── corporate\_config.yaml # Configuration tailored for corporate entities.

│ └── small\_community\_config.yaml # Configuration for small community organizations.

├── workflows/

│ ├── workflow\_rules.yaml # Centralized routing and escalation rules.

│ ├── escalation\_policies.yaml # Centralized escalation policies.

│ └── anonymization\_rules.yaml # Rules for anonymizing sensitive data.

├── logging/

│ ├── logging\_config.yaml # General logging configurations.

│ ├── activity\_log\_settings.yaml # Activity logging configurations.

│ ├── error\_log\_settings.yaml # Error logging configurations.

│ └── security\_log\_settings.yaml # Security logging configurations.

├── email/

│ ├── email\_config.yaml # Email server configurations.

│ ├── smtp\_settings.yaml # SMTP configurations.

│ └── imap\_settings.yaml # IMAP configurations.

├── database/

│ ├── db\_config.yaml # Database connection settings.

│ ├── postgres\_settings.yaml # PostgreSQL-specific configurations.

│ └── sqlite\_settings.yaml # SQLite configurations for offline use.

├── system/

│ ├── general\_settings.yaml # General system settings (e.g., timezone, language).

│ ├── retention\_policies.yaml # Log retention policies.

│ └── notification\_settings.yaml # Notification preferences (email/SMS templates).

/domain\_modules/

├── maintenance/

│ ├── tasks/

│ │ ├── plumbing\_tasks.py

│ │ ├── electrical\_tasks.py

│ │ ├── general\_tasks.py

│ │ └── maintenance\_task\_manager.py

│ ├── templates/

│ │ ├── maintenance\_email.html

│ │ ├── report\_template.html

│ └── rules/

│ └── maintenance\_rules.yaml

├── hr/

│ ├── tasks/

│ │ ├── harassment\_tasks.py

│ │ ├── recruitment\_tasks.py

│ │ └── hr\_task\_manager.py

│ ├── templates/

│ │ ├── harassment\_report.html

│ │ ├── recruitment\_email.html

│ │ └── performance\_review.html

│ └── rules/

│ └── hr\_rules.yaml

└── education/

├── tasks/

│ ├── student\_incident\_tasks.py

│ ├── teacher\_feedback\_tasks.py

│ └── education\_task\_manager.py

├── templates/

│ ├── incident\_report.html

│ ├── feedback\_form.html

│ └── parent\_notification.html

└── rules/

└── education\_rules.yaml

/interfaces/

├── api/

│ ├── endpoints/

│ │ ├── email\_endpoints.py

│ │ ├── workflow\_endpoints.py

│ │ ├── task\_endpoints.py

│ │ └── admin\_endpoints.py

│ ├── serializers/

│ │ ├── email\_serializer.py

│ │ ├── workflow\_serializer.py

│ │ ├── task\_serializer.py

│ │ └── admin\_serializer.py

│ ├── auth/

│ │ ├── token\_manager.py

│ │ └── role\_validator.py

├── admin/

│ ├── dashboard/

│ │ ├── templates/

│ │ │ ├── dashboard.html

│ │ │ ├── escalation\_tracker.html

│ │ │ ├── rule\_editor.html

│ │ │ └── log\_retention.html

│ │ ├── controllers/

│ │ │ ├── task\_controller.py

│ │ │ ├── workflow\_controller.py

│ │ │ └── logging\_controller.py

│ │ ├── static/

│ │ │ ├── admin\_dashboard\_styles.css

│ │ │ ├── admin\_dashboard\_scripts.js

│ │ │ └── icons/

│ │ └── settings.py

├── web/

│ ├── web\_client.py

│ ├── templates/

│ │ ├── login.html

│ │ ├── notification\_center.html

│ │ └── user\_profile.html

│ ├── static/

│ │ ├── web\_client\_styles.css

│ │ ├── web\_client\_scripts.js

│ │ └── images/

│ │ └── logo.png

└── notifications/

├── email\_notifier.py

├── sms\_notifier.py

└── notification\_manager.py

/tests/

├── unit/

│ ├── email/

│ │ ├── test\_subject\_parser.py

│ │ ├── test\_body\_parser.py

│ │ ├── test\_metadata\_extractor.py

│ │ └── test\_attachment\_handler.py

│ ├── workflow/

│ │ ├── test\_rule\_loader.py

│ │ ├── test\_rule\_validator.py

│ │ ├── test\_rule\_executor.py

│ │ └── test\_escalation\_handler.py

│ ├── task/

│ │ ├── test\_scheduler.py

│ │ ├── test\_executor.py

│ │ └── test\_notifier.py

│ ├── database/

│ │ ├── test\_postgres\_connector.py

│ │ ├── test\_sqlite\_connector.py

│ │ └── test\_operations.py

│ └── logging/

│ ├── test\_activity\_logger.py

│ ├── test\_error\_logger.py

│ └── test\_security\_logger.py

├── integration/

│ ├── test\_email\_to\_workflow.py

│ ├── test\_task\_to\_escalation.py

│ ├── test\_task\_lifecycle.py

│ └── test\_database\_sync.py

├── performance/

│ ├── test\_redis\_cache.py

│ ├── test\_rabbitmq\_queues.py

│ ├── test\_email\_handling.py

│ └── test\_bulk\_workflows.py

└── security/

├── test\_rbac.py

├── test\_data\_anonymization.py

├── test\_access\_control.py

└── test\_authentication.py

/logs/

├── workflow/

│ ├── workflow\_activity.log # Logs workflow actions and events.

│ ├── escalation\_tracker.log # Logs task escalations and timelines.

│ └── workflow\_performance.log # Logs performance metrics for workflows.

├── tasks/

│ ├── task\_execution.log # Logs task execution details.

│ ├── task\_notifications.log # Logs notifications sent for tasks.

│ └── overdue\_tasks.log # Tracks overdue tasks and escalations.

├── system/

│ ├── system\_activity.log # Logs general system activity.

│ ├── deployment\_actions.log # Logs deployment and update actions.

│ └── performance\_monitor.log # Logs system resource usage metrics.

├── security/

│ ├── access\_control.log # Tracks login attempts and role violations.

│ ├── data\_anonymization.log # Logs anonymization actions for sensitive workflows.

│ └── alerts.log # Logs security alerts and intrusion attempts.

├── email/

│ ├── email\_incoming.log # Logs metadata for incoming emails.

│ ├── email\_outgoing.log # Logs metadata for outgoing emails.

│ └── email\_errors.log # Tracks errors during email handling.

└── retention\_policies/

├── logs\_retention\_policy.yaml # Centralized log retention policies.

└── compliance\_checklist.yaml # Ensures compliance with data regulations.

/backup/

├── scripts/

│ ├── full\_backup.py # Creates full system backups.

│ ├── incremental\_backup.py # Handles incremental backups.

│ ├── scheduled\_backup.sh # Script to schedule backups using cron.

│ └── verify\_backup.py # Validates backup integrity.

├── recovery/

│ ├── restore\_from\_backup.py # Restores system from backups.

│ ├── conflict\_resolver.py # Resolves conflicts during restoration.

│ └── recovery\_logs.log # Logs recovery actions and outcomes.

└── logs/

├── backup\_log.log # Logs backup activities.

└── recovery\_log.log # Tracks recovery operations.

/monitoring/

├── health\_checks/

│ ├── email\_server\_check.py # Verifies email server connectivity.

│ ├── database\_health\_check.py # Monitors database availability and health.

│ ├── queue\_health\_check.py # Checks Redis and RabbitMQ queues.

│ └── system\_health\_check.py # General system health monitoring.

├── performance\_metrics/

│ ├── redis\_metrics.py # Tracks Redis performance and cache hits.

│ ├── rabbitmq\_metrics.py # Monitors RabbitMQ queue length and latency.

│ ├── cpu\_usage.py # Tracks CPU utilization.

│ ├── memory\_usage.py # Tracks memory usage.

│ └── disk\_usage.py # Monitors disk space and I/O.

└── alerts/

├── alert\_rules.yaml # Defines alert thresholds and conditions.

├── email\_alerts.py # Sends alerts via email.

├── sms\_alerts.py # Sends alerts via SMS.

└── alert\_logs.log # Logs triggered alerts.

/templates/

├── email/

│ ├── task\_notification.html # Template for task notifications.

│ ├── escalation\_notice.html # Template for escalation notices.

│ ├── maintenance\_request.html # Template for maintenance requests.

│ ├── harassment\_report.html # Template for sensitive HR reports.

│ └── incident\_alert.html # Template for incident alerts.

├── reports/

│ ├── maintenance\_summary.html # Template for maintenance task summaries.

│ ├── performance\_review.html # Template for employee performance reviews.

│ ├── incident\_report.html # Template for incident reporting.

│ └── feedback\_summary.html # Template for summarizing user feedback.

├── dashboards/

│ ├── admin\_dashboard.html # Template for the admin dashboard.

│ ├── escalation\_tracker.html # Dashboard for tracking escalations.

│ ├── workflow\_editor.html # Template for workflow editing.

│ └── analytics\_overview.html # Template for performance analytics.

├── notifications/

│ ├── success\_message.html # Template for success notifications.

│ ├── error\_message.html # Template for error notifications.

│ └── escalation\_reminder.html # Template for escalation reminders.

└── forms/

├── login\_form.html # Template for login forms.

├── registration\_form.html # Template for user registration.

├── feedback\_form.html # Template for collecting feedback.

└── report\_issue\_form.html # Template for reporting issues.

Here’s a more technical and machine-friendly annex to enhance the **use of the Orgo v2 file system for programming**. This document is intended to help developers and AI systems interact effectively with the file structure by clarifying its architecture, relationships, and implementation principles.

# **Technical Annex: Programming Guidelines for Orgo v2 File Structure**

This document provides technical details and programming-focused insights for interacting with the **Orgo v2 General File Structure**. It is designed to ensure that AI systems and developers understand how to leverage the file system to build, extend, and maintain the platform effectively.

## **1. Programming Purpose of Each Directory**

### **/core\_services/**

* **Purpose**: Provides reusable components that execute fundamental operations such as parsing, rule application, database interactions, task scheduling, and logging.
* **Programming Notes**:
  + Modules must be stateless where possible to support scalability.
  + All services depend on external configurations from /config/ and integrate with logs in /logs/.
  + Each service exposes standardized functions or classes (e.g., TaskScheduler, DatabaseConnector) for predictable interaction.

### **/config/**

* **Purpose**: Centralizes dynamic configurations that define workflows, logging behavior, database connections, and organization-specific rules.
* **Programming Notes**:
  + Configuration files are expected in YAML or JSON format.
  + Standard keys (e.g., escalation\_policies, notification\_settings) must be used across all configuration files to ensure interoperability.
  + When adding new configurations, ensure backward compatibility by including defaults in default\_config.yaml.

### **/domain\_modules/**

* **Purpose**: Encapsulates workflows, task definitions, templates, and rules tailored to specific domains such as maintenance, HR, or education.
* **Programming Notes**:
  + Follow a strict file naming convention (e.g., <domain\_name>\_tasks.py, <domain\_name>\_rules.yaml) to maintain consistency.
  + Modules must adhere to the input/output schema defined by /core\_services/ for seamless integration.
  + Templates should use placeholders for dynamic values, compatible with the template engine configured in /interfaces/.

### **/interfaces/**

* **Purpose**: Manages user-facing operations, including API endpoints, dashboards, and notifications.
* **Programming Notes**:
  + API endpoints in /interfaces/api/ must follow REST principles and use a consistent naming pattern (e.g., get\_<resource>, update\_<resource>).
  + All user-facing templates in /interfaces/web/templates/ should reference shared styles and scripts from /interfaces/web/static/ for consistency.
  + Notifications rely on preferences defined in /config/organizations/, so changes to notification logic should account for these configurations.

### **/logs/**

* **Purpose**: Captures structured logs for workflows, tasks, system activity, and security events.
* **Programming Notes**:
  + Use a uniform logging format (e.g., JSON with keys like timestamp, level, message, module) for all log entries.
  + Implement role-based access controls (RBAC) when accessing sensitive logs like security/access\_control.log.
  + Ensure logs comply with retention policies defined in /config/logging/.

### **/infrastructure/**

* **Purpose**: Supports deployment, monitoring, backups, synchronization, and system scalability.
* **Programming Notes**:
  + Deployment scripts (e.g., docker-compose.yaml, setup.py) must reference environment-specific configurations in /config/.
  + Monitoring tools in /infrastructure/monitoring/ should interact with logs in /logs/ and external visualization systems (e.g., Prometheus).
  + Synchronization scripts must resolve conflicts programmatically using timestamp-based reconciliation.

### **/tests/**

* **Purpose**: Ensures functionality, reliability, and performance through comprehensive testing.
* **Programming Notes**:
  + Unit tests in /tests/unit/ must mock dependencies to isolate logic.
  + Integration tests in /tests/integration/ must validate cross-module interactions.
  + Performance tests in /tests/performance/ should simulate high-load scenarios using Redis and RabbitMQ.

## **2. Interaction Principles**

### **Centralized Configuration Usage**

All modules must query /config/ for dynamic settings:

* Use loaders (e.g., yaml.safe\_load) to read configurations into objects or dictionaries.
* Example: A workflow task may load its routing rules dynamically:

import yaml

with open('/config/workflows/workflow\_rules.yaml', 'r') as file:

rules = yaml.safe\_load(file)

### **Module Integration**

Modules must use clearly defined interfaces to communicate:

* Core services provide reusable APIs (e.g., DatabaseConnector.query()).
* Domain modules invoke these APIs and transform their outputs to user-facing formats.

### **Template Standardization**

Templates in /domain\_modules/templates/ and /interfaces/web/templates/ must follow a shared syntax for placeholders to ensure compatibility with the rendering engine:

* Use double curly braces for placeholders (e.g., {{ variable\_name }}).

## **3. Programming Practices for Scalability**

### **Task Queues**

* Use Redis for caching and RabbitMQ for task queues, integrating them into /core\_services/task/.
* Ensure message durability and acknowledgment to handle high-load scenarios.

### **Stateless Design**

* All workflows and tasks should rely on database state rather than maintaining in-memory state for scalability in distributed environments.

### **Offline Support**

* Implement offline synchronization logic in /infrastructure/scripts/ to ensure seamless operations during network outages:
  + Reconcile SQLite and PostgreSQL data using timestamp comparison.

## **4. Implementation Constraints**

1. **File Naming Conventions**
   * Ensure unique, descriptive file names (e.g., workflow\_manager.py, notification\_settings.yaml) to avoid conflicts.
2. **Directory Depth**
   * Keep directories shallow and modular to maintain clarity and facilitate navigation.
3. **Backward Compatibility**
   * Changes to configuration keys or module interfaces must preserve compatibility with existing workflows.
4. **Security Enforcement**
   * Sensitive operations (e.g., logging, data synchronization) must validate user roles and permissions dynamically.

## **5. Dynamic Adaptability**

The file system is designed to adapt dynamically to new requirements:

* **Adding New Domain Modules**:
  + Create a new directory in /domain\_modules/ with the same structure as existing modules.
  + Reference shared rules and templates from /config/ and /templates/ for consistency.
* **Extending Workflows**:
  + Add new rules to /config/workflows/workflow\_rules.yaml.
  + Update loaders in /core\_services/workflow/ to handle new conditions dynamically.

## **6. Error Handling and Debugging**

* **Error Logging**:
  + Use /logs/system/ for capturing unexpected errors across all modules.
  + Tag log entries with identifiers (e.g., module\_name) to facilitate debugging.
* **Testing Coverage**:
  + Ensure every module has corresponding unit and integration tests in /tests/.

## **Conclusion**

This annex provides a detailed, programming-friendly guide to the **Orgo v2 file system**, ensuring its elements are usable and extensible by AI systems and developers. By adhering to these principles, the system will remain robust, scalable, and adaptable to evolving requirements.