Bansuri

Release 1.0

Bansuri Contributors

Jan 22, 2026

START HERE

Task Orchestration & Management System

Bansuri is a flexible, easy-to-use system for running and monitoring multiple scripts with configurable scheduling, timeouts, retries, and notifications.

What You Can Do:

- Run shell commands on timers or cron schedules
- Retry failed tasks automatically
- Monitor task execution with detailed logs
- Send alerts on failures
- Manage multiple tasks from one configuration file

What's Coming: Task dependencies, user switching, hot-reload, and more.

See Feature Status for implementation details.

CHAPTER

ONE

QUICK START GUIDE

Get Bansuri running in 5 minutes.

1.1 Prerequisites

- Python 3.8 or higher
- `pip` package manager

1.2 Installation

```
# Clone the repository
git clone https://github.com/aziabatz/bansuri.git
cd bansuri

# Install in development mode
pip install -e .

# Verify installation
bansuri --help
```

1.3 Your First Task

Step 1: Create `scripts.json`:

```
{
  "version": "1.0",
  "scripts": [
    {
      "name": "hello-world",
      "command": "echo 'Hello from Bansuri!'",
      "timer": "30",
      "description": "Prints a greeting every 30 seconds"
    }
  ]
}
```

Step 2: Run Bansuri:

```
bansuri
```

Expected Output:

```
[2026-01-19 10:00:00] [MASTER] ========================================
[2026-01-19 10:00:00] [MASTER] BANSURI ORCHESTRATOR STARTED
[2026-01-19 10:00:00] [MASTER] ========================================
[2026-01-19 10:00:00] [hello-world] Runner started.
[2026-01-19 10:00:01] [hello-world] Timer configured: running every 30s (30s)
[2026-01-19 10:00:01] [hello-world] Executing shell command: echo 'Hello from Bansuri!'
[2026-01-19 10:00:01] [hello-world] Process finished with code 0
[2026-01-19 10:00:31] [hello-world] Executing shell command: echo 'Hello from Bansuri!'
```

Step 3: Stop with `Ctrl+C`

1.4 What Just Happened?

1. Bansuri read `scripts.json`
2. Found task `hello-world`
3. Started a `TaskRunner` thread for the task
4. Task ran immediately, then every 30 seconds
5. Logs showed each execution

1.4.1 Useful Examples

1.5 Backup Database Every Hour

Add this to your `scripts.json`:

```json
{
  "name": "hourly-backup",
  "command": "pg_dump mydb | gzip > /backups/db-$(date +%Y%m%d_%H%M%S).sql.gz",
  "timer": "3600",
  "timeout": "15m",
  "on-fail": "restart",
  "times": 2,
  "working-directory": "/backups",
  "description": "Backs up PostgreSQL database hourly"
}
```

1.6 Run Health Checks Every Minute

```json
{
  "name": "health-check",
  "command": "/usr/local/bin/health-check.sh",
  "timer": "60",
  "timeout": "30s",
  "on-fail": "stop",
  "stdout": "/var/log/healthcheck.log",
```

(continues on next page)

(continued from previous page)

```
  "stderr": "combined",
  "description": "System health check every minute"
}
```

1.7 Data Sync with Retries

```
{
  "name": "sync-data",
  "command": "rsync -av /data/source /data/backup",
  "timer": "300",
  "on-fail": "restart",
  "times": 3,
  "timeout": "5m",
  "notify": "mail",
  "description": "Sync data every 5 minutes with retry"
}
```

1.8 Complete Beginner Configuration

```
{
  "version": "1.0",
  "notify_command": "mail -s '[Bansuri] Task Failed' admin@example.com",
  "scripts": [
    {
      "name": "task-1-quick",
      "command": "echo 'Quick task'",
      "timer": "60",
      "description": "Runs every minute"
    },
    {
      "name": "task-2-medium",
      "command": "/opt/scripts/process.sh",
      "timer": "300",
      "timeout": "2m",
      "on-fail": "restart",
      "times": 2,
      "description": "Runs every 5 minutes with retry"
    },
    {
      "name": "task-3-critical",
      "command": "/opt/scripts/critical-job.sh",
      "timer": "3600",
      "timeout": "30m",
      "on-fail": "restart",
      "times": 3,
      "notify": "mail",
      "stdout": "/var/log/critical.log",
      "stderr": "combined",
      "description": "Critical hourly job with monitoring"
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

1.8.1 Common Tasks

Run once and exit:

```json
{
  "name": "one-time-task",
  "command": "setup.sh"
}
```

Run periodically:

```json
{
  "name": "periodic-task",
  "command": "cleanup.sh",
  "timer": "86400"
}
```

Run with timeout:

```json
{
  "name": "timeout-task",
  "command": "long-running.sh",
  "timeout": "30m"
}
```

Run with retries:

```json
{
  "name": "retry-task",
  "command": "api-call.py",
  "on-fail": "restart",
  "times": 3
}
```

Run with logging:

```json
{
  "name": "logged-task",
  "command": "data-process.sh",
  "stdout": "/var/log/process.log",
  "stderr": "combined"
}
```

Run with notifications:

```json
{
  "name": "monitored-task",
  "command": "critical-check.sh",
  "notify": "mail",
```

(continues on next page)

(continued from previous page)

```
  "on-fail": "restart",
  "times": 2
}
```

1.8.2 Verify Your Configuration

Before running:

```
# Validate JSON
python -m json.tool scripts.json

# Validate with Bansuri
python -c "
from bansuri.base.config_manager import BansuriConfig
config = BansuriConfig.load_from_file('scripts.json')
print('✓ Configuration valid')
for script in config.scripts:
    print(f'  - {script.name}')
"
```

1.8.3 Next Steps

- Read *Configuration* for advanced options
- Set up *Notifications* for alerts
- Deploy to production with *Deployment Guide*
- See troubleshooting-detailed if issues arise
- Create custom tasks with *Creating Custom Tasks*

1.8.4 Tips and Tricks

Use environment variables:

```
export BACKUP_DIR=/backups
bansuri
```

Then in scripts.json:

```
{
  "command": "pg_dump mydb > $BACKUP_DIR/db.sql"
}
```

Use cron-like schedules as timer:

```
{
  "timer": "3600"
}
```

Conversion guide: - `*/5 * * * *` (every 5 minutes) → `"timer": "300"` - `0 * * * *` (hourly) → `"timer": "3600"` - `0 0 * * *` (daily) → `"timer": "86400"`

Test commands before adding:

```
# Run the command manually first
/usr/local/bin/my-script.sh
echo $?  # Check exit code
```

Redirect verbose output:

```
{
  "command": "./verbose-script.sh 2>/dev/null",
  "stdout": "/var/log/task.log"
}
```

1.8.5 Troubleshooting

Task not running?

1. Check it's in the config: `grep "name" scripts.json`
2. Verify the command exists: `which command_name`
3. Test manually: `/path/to/command`
4. Check permissions: `ls -la /path/to/command`

See troubleshooting-detailed for more help.

1.8.6 Getting Support

- Full documentation: See the other RST files
- Report issues: GitHub Issues
- Discuss: GitHub Discussions

Happy orchestrating!

CHAPTER

TWO

INSTALLATION

Install Bansuri and its optional dependencies.

2.1 Requirements

- Python 3.8 or higher
- `pip` package manager
- `git` (for source installation)

2.1.1 Installation Methods

2.2 From Source (Recommended for Development)

```
git clone https://github.com/aziabatz/bansuri.git
cd bansuri
pip install -e .
```

Verify installation:

```
bansuri --help
```

2.2.1 Optional Dependencies

For Cron Scheduling:

```
pip install croniter
```

This enables `schedule-cron` field in tasks.

For Email Notifications:

You need system `mail` command (usually pre-installed):

```
# Ubuntu/Debian
sudo apt-get install mailutils

# CentOS/RHEL
sudo yum install mailx

# macOS
# Usually pre-installed
```

For Python Script Tasks (Future):

Install any dependencies your Python scripts need.

2.2.2 Verify Installation

Check that Bansuri is installed correctly:

```
python -c "import bansuri; print(bansuri.__version__)"
```

Expected output:

```
0.1.0
```

2.2.3 Next Steps

1. Read the *Quick Start Guide* guide
2. Check *Configuration* documentation
3. Explore *Reference* for all options
4. See *Troubleshooting* if issues arise

CHAPTER

THREE

CONCEPTS AND TERMINOLOGY

3.1 Core Components

3.1.1 Task

A **task** is a unit of work to be executed by Bansuri.

A task consists of:

- **Name**: Unique identifier
- **Command**: What to execute (shell command)
- **Schedule**: When/how often to run (timer, cron, or once)
- **Policies**: Timeout, retries, notifications

Example:

```
{
  "name": "backup-database",
  "command": "pg_dump production | gzip > backup.sql.gz",
  "timer": "86400",
  "timeout": "1h",
  "on-fail": "restart",
  "notify": "mail"
}
```

3.1.2 Orchestrator

The **Orchestrator** is the main coordinator. It:

- Reads configuration from `scripts.json`
- Creates `TaskRunner` for each task
- Monitors all tasks continuously
- Handles graceful shutdown (SIGTERM/SIGINT)
- Syncs configuration changes

The Orchestrator runs continuously in a single process.

3.1.3 TaskRunner

Each task gets its own **TaskRunner** that:

- Runs in a dedicated thread
- Spawns the task process
- Manages timeouts and retries
- Captures and redirects output
- Sends notifications on failure

3.2 Execution Models

Bansuri supports different scheduling modes:

One-Time Execution

Task runs once at startup:

```
{
  "name": "initialize",
  "command": "python init.py"
}
```

Timer-Based (Fixed Interval)

Task runs repeatedly at fixed intervals:

```
{
  "name": "monitor",
  "command": "python health_check.py",
  "timer": "300"
}
```

Supported formats: - `"60"` → every 60 seconds - `"5m"` → every 5 minutes - `"1h"` → every hour

Cron-Based (Scheduled Times)

Task runs on cron schedule (requires `croniter`):

```
{
  "name": "daily-backup",
  "command": "bash backup.sh",
  "schedule-cron": "0 2 * * *"
}
```

Cron format: `"minute hour day month weekday"`

3.3 Execution Control

Failure Handling

When a task fails (non-zero exit code):

- `on-fail: "stop"` → Stop and don't retry (default)
- `on-fail: "restart"` → Retry immediately

Combined with `times` for max retries:

```json
{
  "name": "api-sync",
  "command": "python sync.py",
  "on-fail": "restart",
  "times": 3
}
```

Timeout Management

Kill task if it runs too long:

```json
{
  "name": "long-task",
  "command": "bash slow.sh",
  "timeout": "30m"
}
```

Supported formats: - `"30s"` → 30 seconds - `"5m"` → 5 minutes - `"1h"` → 1 hour

Success Definition

By default, only exit code 0 is success. Override with:

```json
{
  "name": "exit-handler",
  "command": "bash script.sh",
  "success-codes": [0, 1, 2]
}
```

Exit codes 1 and 2 won't trigger retries or notifications.

3.4 Task Lifecycle

A task goes through these states:

1. **PENDING** → Task queued, waiting to start
2. **RUNNING** → Process is executing
3. **TIMEOUT** → Task exceeded timeout limit
4. **SUCCESS** → Task completed with success code
5. **FAILED** → Task exited with non-success code
6. **RESTARTING** → Task failed, retrying (if `on-fail:  restart`)
7. **STOPPED** → Task stopped by user or reached max attempts

3.5 Configuration Synchronization

The Orchestrator monitors `scripts.json` for changes:

- **New tasks** → Automatically started
- **Deleted tasks** → Automatically stopped
- **Modified tasks** → NOT YET (requires hot-reload)

See ../NOT_IMPLEMENTED.md for feature status.

```
{
  "name": "monitor",
  "command": "check.sh",
  "timer": "300"
}
```

Cron-based Execution (NOT IMPLEMENTED)

Task runs on cron schedule:

```
{
  "name": "nightly-job",
  "command": "backup.sh",
  "schedule-cron": "0 2 * * *"
}
```

3.6 Configuration Hierarchy

Global Configuration (scripts.json root level)

```
{
  "version": "1.0",
  "notify_command": "mail ..."
}
```

Per-task Configuration (scripts[].* level)

```
{
  "name": "task-name",
  "command": "...",
  "timer": "300",
  "timeout": "30s",
  "on-fail": "restart",
  ...
}
```

3.7 Task Lifecycle

A task progresses through these states:

1. **PENDING**: Registered in config, waiting to run
2. **RUNNING**: Process spawned with active PID
3. **SUCCESS**: Completed with success code (default: 0)
4. **FAILED**: Exited with non-zero code
5. **TIMEOUT**: Killed due to exceeding timeout
6. **RESTARTING**: Retrying after failure (if on-fail=restart)
7. **STOPPED**: Removed from config or shutdown

3.7.1 State Transitions

```
PENDING
  ↓
RUNNING → SUCCESS (end)
  ↓
FAILED
  ↓
on-fail=stop? → STOPPED (end)
      |
      ↓ on-fail=restart?
RESTARTING
  ↓
(back to RUNNING)
```

3.8 Failure Policies

on-fail: "stop" (default)

Task stops immediately on failure:

```
RUNNING → FAILED → STOPPED
```

on-fail: "restart"

Task restarts up to `times` attempts:

```
RUNNING → FAILED → RESTARTING → RUNNING → FAILED → ... (up to 'times')

If max attempts reached:
FAILED → STOPPED
```

Example:

```
{
  "on-fail": "restart",
  "times": 3
}
```

3.8.1 Timeout Behavior

When timeout is exceeded:

```
RUNNING → (timeout reached) → FAILED → (apply on-fail policy)
```

The process is forcefully terminated with SIGKILL.

3.8.2 Success Codes

By default, only exit code `0` is considered success. You can define custom success codes:

```
{
  "success-codes": [0, 2, 137]
}
```

Exit codes not in this list are considered failures.

3.9 Output Management

Bansuri redirects task output according to configuration:

No redirection

Output goes to console (Bansuri's stdout/stderr):

```
{}
```

```

**File redirection**

```
{
  "stdout": "/var/log/task.log",
  "stderr": "combined"
}
```

**Separate files**

```
{
  "stdout": "/var/log/task.out",
  "stderr": "/var/log/task.err"
}
```

Combined output

Both stdout and stderr go to the same file:

```
{
  "stderr": "combined"
}
```

3.10 Resource Management

Working Directory

Tasks execute in the specified directory. Affects:

- Relative paths in commands
- File operations
- Working directory of child processes

Timeout

Maximum time a task can run. Process killed if exceeded.

Units: - Seconds: `"30"` or `30` - Seconds suffix: `"30s"` - Minutes: `"5m"` = 300 seconds - Hours: `"2h"` = 7200 seconds

Process Groups

Each task runs in its own process group. Allows:

- Killing task and all children
```

- Resource isolation
- Clean shutdown

3.11 Notification System

Triggered on task failure when `notify: "mail"` is set.

Components:

- **FailureInfo**: Data structure with failure details
- **Notifier**: Abstract base class for notification handlers
- **CommandNotifier**: Sends notification via shell command

Failure notification includes:

- Task name and command
- Return code and exit details
- Attempt number
- Timestamp
- Stdout/stderr output
- Task description

See *Notifications* for details.

3.12 Abstract Tasks

Custom task implementations implement the `AbstractTask` interface:

```
class AbstractTask(ABC):
    @abstractmethod
    def run(self) -> int:
        """Execute task, return exit code"""
        pass

    @abstractmethod
    def stop(self) -> None:
        """Stop task if running"""
        pass
```

Benefits:

- Complex business logic
- Native Python integration
- Stateful operations
- Resource cleanup

See *Creating Custom Tasks* for examples.

3.13 Configuration Management

Loading

Configuration loaded from `scripts.json` when Bansuri starts:

1. Parse JSON file
2. Validate schema
3. Create `BansuriConfig` object
4. Spawn `TaskRunner` for each script

Validation

Each script is validated for:

- Required fields (name, command)
- Valid field types
- Sensible values
- Scheduling requirements

Synchronization

Bansuri periodically syncs config changes:

1. Detect added tasks → spawn `TaskRunner`
2. Detect removed tasks → stop and remove `TaskRunner`
3. Detect modified tasks → restart `TaskRunner`

3.14 Performance Concepts

Concurrency

Each task runs in its own thread. Multiple tasks execute concurrently.

I/O-bound tasks (network, disk): - Can run many concurrently - Limited by system I/O, not CPU

CPU-bound tasks (computation): - Limited by CPU cores - Too many concurrent = resource contention

Resource Limits

Monitor and limit:

- Memory: Each process and total system
- CPU: Per-process or total
- File descriptors
- Disk I/O

3.15 Observability Concepts

Logging

Structured logging from:

- Orchestrator: Task lifecycle events

- TaskRunner: Execution events
- Tasks: Custom logs

Monitoring

Observable metrics:

- Task execution count
- Success/failure rates
- Execution time
- Resource usage
- Timeout occurrences

Alerting

Based on:

- Task failures
- Notification triggers
- System resource issues
- Configuration changes

3.16 Security Concepts

Process Isolation

Each task runs in separate process with:

- Independent memory space
- Own file descriptors
- Own process group

User Context

(NOT IMPLEMENTED) Tasks can run as specific users with:

- Reduced privileges
- Different permissions
- Isolated home directories

Working Directory

Controls where tasks execute and what files they access.

Permissions

- Script execute permissions
- Log directory write permissions
- Output file creation permissions

3.17 High Availability Concepts

Restart Policies

Automatic recovery from failures:

- `on-fail: "restart"`
- `times: N` attempts
- Configurable retry behavior

Health Checks

Monitor task execution:

- Track exit codes
- Detect hangs
- Measure latency

Graceful Shutdown

Clean termination:

1. Send SIGTERM to tasks
2. Wait for graceful shutdown
3. Force SIGKILL if needed
4. Flush logs and state

3.18 Scaling Concepts

Single Instance

Suitable for:

- < 100 tasks
- < 1000 executions/minute
- Single machine
- Non-critical workloads

Multiple Instances

For larger deployments:

- Partition tasks across instances
- Share configuration via version control
- Centralize logging
- Coordinate via monitoring

Containerized Deployment

Run in:

- Docker containers
- Kubernetes pods
- Cloud container services

Benefits:

- Reproducible environment
- Easy scaling
- Simplified deployment
- Resource limits

3.19 Related Concepts

Cron (traditional Linux scheduling)

Similar to Bansuri's timer-based execution but:

- System-level scheduler
- Separate from process lifecycle
- Limited monitoring

Bansuri advantages:

- Process-level control
- In-memory state
- Unified logging
- Flexible retry logic

Process Managers (supervisord, systemd)

Similar in goal but different scope:

- Process managers: Keep services running
- Bansuri: Orchestrate task execution

Bansuri can run under process managers for higher availability.

3.20 Terminology Reference

Term	Definition
Orchestrator	Central coordinator managing all tasks
TaskRunner	Thread-based executor for a single task
Task	Unit of work to be executed
Script	Configuration entry for a task
Configuration	JSON file defining tasks and behavior
Exit Code	Process return value (0=success, non-zero=failure)
Timeout	Maximum execution time for a task
Failure Policy	What to do when task fails (stop or restart)
Notification	Alert sent when task fails
FailureInfo	Data structure with task failure details
Process Group	Group of processes from same task
Working Directory	Directory where task executes
Success Codes	Exit codes considered successful

CHAPTER

FOUR

SYSTEM ARCHITECTURE

Bansuri is designed as a modular task orchestration system. It separates the concerns of task definition, scheduling, and execution monitoring.

4.1 High-Level Overview

The system consists of two main components:

1. **Orchestrator**: Reads the configuration from `scripts.json`, manages the lifecycle of all tasks, synchronizes configuration changes, and coordinates task execution.
2. **TaskRunner**: Encapsulates the logic for spawning a subprocess, managing a control thread, capturing output, and handling lifecycle events (start, stop, timeout, restarts).

Note

Bansuri uses Python's `subprocess` module for process isolation, ensuring that a crashing script does not bring down the main orchestrator. Each task runs in its own thread and process.

4.2 Execution Models

Bansuri supports multiple execution models through `TaskRunner`:

- **Simple Execution**: Run the task once and stop.
- **Timer-based Execution**: Run the task at fixed intervals (configured via `timer` parameter).
- **Cron-based Execution**: Run the task on a schedule (configured via `schedule-cron`). NOT IMPLEMENTED.

4.3 Task Lifecycle

A task goes through several states during its execution:

- **PENDING**: The task is about to start (in the execution loop).
- **RUNNING**: The process has been spawned and has an active PID.
- **COMPLETED**: The process exited with a success code (default: 0).
- **FAILED**: The process exited with a non-zero return code or was killed.
- **TIMED_OUT**: The TaskRunner killed the process because it exceeded its `timeout` limit.

4.4 Error Handling and Restart Policies

If a task fails, the **on-fail** policy kicks in:

- **"stop"**: Task execution stops immediately on failure.
- **"restart"**: Task is restarted up to `times` attempts with a brief delay between attempts.

The configuration in *Configuration* determines the behavior on task failure.

4.5 Output Management

Bansuri redirects stdout and stderr according to the task configuration:

- **"combined"** (default): Both stdout and stderr are written to the same file.
- **File path**: Stdout/stderr are redirected to the specified file.
- **None**: Output is not captured.

4.6 Extensibility

Bansuri is built to be extended. You can create custom task implementations by inheriting from the `AbstractTask` class and implementing the `run()` and `stop()` methods. Tasks implementing this interface can be loaded and executed by Bansuri with the `no-interface` configuration option.

4.7 Notification System

Bansuri includes a notification system that can alert you when tasks fail. The system is extensible and supports:

- **Command-based notifications**: Execute any shell command to send alerts (email, Slack, webhooks, etc.)
- **Custom notifiers**: Extend the `Notifier` base class to implement custom notification handlers

For detailed information, see *Notifications*.

CHAPTER

FIVE

CONFIGURATION

How to configure tasks in Bansuri.

5.1 Configuration File

Bansuri uses **JSON** configuration files to define tasks.

Default Location: `scripts.json` in current directory

Example:

```
{
  "version": "1.0",
  "notify_command": "mail -s 'Alert' admin@example.com",
  "scripts": [
    {
      "name": "task-1",
      "command": "python script.py",
      "timer": "300"
    }
  ]
}
```

5.1.1 Minimal Task

Only **name** and **command** are required:

```
{
  "version": "1.0",
  "scripts": [
    {
      "name": "my-task",
      "command": "bash do_something.sh"
    }
  ]
}
```

5.2 Task Parameters Reference

Scheduling (pick one):

Execution Control:

Output & Logs:

Parameter	Example	Description
`stdout`	`"task.log"`	File to save stdout
`stderr`	`"combined"`	File for stderr or "combined" (default: combined)
`working-directory`	`"/app/scripts"`	Directory to run command in
`description`	`"Daily backup"`	Human-readable description

Advanced (not yet implemented):

Parameter	Example	Description
`depends-on`	`["task-1"]`	Run after other tasks complete
`user`	`"postgres"`	Run as different user
`priority`	`10`	Process priority (nice value)
`environment-file`	`"/etc/env.json"`	Load environment variables from file

5.2.1 Time Format Examples

```
{
  "timer": "30",       // Every 30 seconds
  "timer": "5m",       // Every 5 minutes
  "timer": "1h",       // Every hour
  "timeout": "60s",    // Timeout after 60 seconds
  "timeout": "30m",    // Timeout after 30 minutes
  "schedule-cron": "*/5 * * * *"    // Every 5 minutes (cron)
}
```

5.2.2 Complete Example

```
{
  "version": "1.0",
  "notify_command": "mail -s 'Bansuri: {task}' admin@example.com",
  "scripts": [
    {
      "name": "database-backup",
      "command": "pg_dump mydb | gzip > backup.sql.gz",
      "working-directory": "/backups",
      "timer": "86400",
      "timeout": "1h",
      "times": 2,
      "on-fail": "restart",
      "stdout": "backup.log",
      "stderr": "combined",
      "notify": "mail",
```

(continues on next page)

(continued from previous page)

```
      "success-codes": [0],
      "description": "Daily PostgreSQL backup"
    }
  ]
}
```

See *Reference* for more examples and *Advanced Configuration* for complex setups.

CHAPTER

SIX

REFERENCE

Complete reference for configuration and command-line usage.

6.1 Quick Configuration Examples

6.1.1 Minimal Task

```json
{
  "version": "1.0",
  "scripts": [
    {
      "name": "my-task",
      "command": "echo 'hello'"
    }
  ]
}
```

Result: Task runs once at startup.

—

6.1.2 Timer-Based Task (Every 5 Minutes)

```json
{
  "name": "monitor",
  "command": "python monitor.py",
  "timer": "300"
}
```

—

6.1.3 Cron-Scheduled Task (Daily at 2 AM)

```json
{
  "name": "backup",
  "command": "bash backup.sh",
  "schedule-cron": "0 2 * * *"
}
```

Requirements: `pip install croniter`

—

6.1.4 Task with Retry Logic

```
{
  "name": "api-call",
  "command": "curl -f https://api.example.com/sync",
  "timer": "3600",
  "timeout": "30s",
  "on-fail": "restart",
  "times": 3,
  "success-codes": [0]
}
```

Behavior: Runs hourly, retries 3 times on failure, kills if > 30s.

—

6.1.5 Complete Task Example

```
{
  "version": "1.0",
  "notify_command": "mail -s 'Alert: {task}' admin@example.com",
  "scripts": [
    {
      "name": "database-maintenance",
      "command": "python /app/maintenance.py",
      "working-directory": "/app",
      "timer": "86400",
      "timeout": "2h",
      "times": 2,
      "on-fail": "restart",
      "stdout": "/var/log/maintenance.log",
      "stderr": "combined",
      "notify": "mail",
      "success-codes": [0],
      "description": "Daily database maintenance"
    }
  ]
}
```

6.2 Parameter Reference

6.2.1 Required Parameters

Parameter	Description
`version`	Config version (usually “1.0”)
`scripts`	Array of task definitions
`name` (per-task)	Unique task identifier
`command`	Command or script to execute

6.2.2 Optional Scheduling

Parameter	Description
`timer`	Interval in seconds (e.g., "300", "5m")
`schedule-cron`	Cron expression (e.g., "0 2 * * *")

Note: Pick ONE scheduling method. If none specified, runs once at startup.

6.2.3 Optional Execution Control

Parameter	Default	Description
`timeout`	None	Max execution time (e.g., "5m")
`on-fail`	"stop"	Behavior: "stop" or "restart"
`times`	1	Max attempts
`success-codes`	[0]	Acceptable exit codes (array)

6.2.4 Optional Output

Parameter	Default	Description
`stdout`	None	File to save stdout
`stderr`	"combined"	File for stderr or "combined"
`working-directory`	None	Directory to run command in
`description`	""	Human-readable description

6.3 Time Format Reference

All time-based parameters support flexible formats:

Format	Means
"30"	30 seconds
"30s"	30 seconds
"5m"	5 minutes (= 300 seconds)
"1h"	1 hour (= 3600 seconds)
"86400"	1 day in seconds

6.3.1 Common Intervals

Interval	Timer Value	Alternative Formats
Every 30 seconds	"30"	"30s"
Every minute	"60"	"1m"
Every 5 minutes	"300"	"5m"
Every hour	"3600"	"1h"
Every day	"86400"	"24h"

6.4 Command-Line Reference

6.4.1 Start Bansuri

```
# Use default scripts.json
bansuri

# Use custom config file
bansuri --config /path/to/config.json

# Show help
bansuri --help
```

6.4.2 Validation

Check if config file is valid JSON:

```
python -m json.tool scripts.json
```

Quick Validation Script:

```
python -c "
from bansuri.base.config_manager import BansuriConfig
try:
    config = BansuriConfig.load_from_file('scripts.json')
    print(' Configuration is valid')
    for task in config.scripts:
        print(f'   - {task.name}')
except Exception as e:
    print(f' Error: {e}')
"
```

6.4.3 Testing Commands

Before adding to Bansuri, test your command:

```
# Test shell command
bash -x /path/to/script.sh

# Check Python syntax
python -m py_compile my_script.py

# Run with timeout
timeout 30s python my_script.py
```

6.5 Common Mistakes

1. **Missing croniter library**

   ```
   ERROR: 'croniter' library is missing
   ```

   Fix:

```
pip install croniter
```

2. **Invalid JSON**

   Symptom: Task doesn't start, no error message

   Fix:

```
python -m json.tool scripts.json
```

3. **Command not found**

   Symptom: Exit code 127

   Fix: Use absolute path to script

```
"command": "/usr/local/bin/my_script.sh"
```

4. **Permission denied**

   Symptom: Exit code 126

   Fix: Make script executable

```
chmod +x /path/to/script.sh
```

See *Troubleshooting* for more help.

6.5.1 View Logs

For systemd:

```
# Live logs
journalctl -u bansuri -f

# Last 50 lines
journalctl -u bansuri -n 50

# Since specific time
journalctl -u bansuri --since "2024-01-19 10:00:00"

# Only errors
journalctl -u bansuri -p err
```

For file logging:

```
# Live logs
tail -f /var/log/bansuri.log

# Last 100 lines
tail -100 /var/log/bansuri.log

# Follow and grep
tail -f /var/log/bansuri.log | grep FAILED
```

6.5.2 System Management

```
# Start
systemctl start bansuri

# Stop
systemctl stop bansuri

# Restart
systemctl restart bansuri

# Status
systemctl status bansuri

# Enable on boot
systemctl enable bansuri

# Disable on boot
systemctl disable bansuri
```

6.5.3 Validation

Validate configuration syntax:

```
python -m json.tool scripts.json
```

Validate with Bansuri:

```
python -c "
from bansuri.base.config_manager import BansuriConfig
try:
    config = BansuriConfig.load_from_file('scripts.json')
    print('✓ Valid')
except Exception as e:
    print(f' Error: {e}')
"
```

6.5.4 Testing

Test a command before adding to config:

```
bash -x /path/to/script.sh
```

Profile execution time:

```
time /path/to/script.sh
```

Monitor in real-time:

```
watch -n 1 'ps aux | grep python'
```

6.5.5 Environment Variables

Variable	Default	Description
BANSURI_ENV	production	Environment (dev/staging/production)
BANSURI_LOG_LEVEL	INFO	Logging verbosity (DEBUG/INFO/WARNING/ERROR)

Set before running:

```
export BANSURI_LOG_LEVEL=DEBUG
bansuri
```

6.6 Configuration Examples

6.6.1 Example 1: Simple Health Check

```
{
  "version": "1.0",
  "scripts": [
    {
      "name": "health-check",
      "command": "curl -f https://example.com/health",
      "timer": "60",
      "timeout": "10s"
    }
  ]
}
```

6.6.2 Example 2: Database Backup with Retry

```
{
  "version": "1.0",
  "notify_command": "mail -s 'Backup Alert' ops@example.com",
  "scripts": [
    {
      "name": "backup-db",
      "command": "pg_dump mydb | gzip > backup.sql.gz",
      "timer": "86400",
      "timeout": "2h",
      "on-fail": "restart",
      "times": 3,
      "notify": "mail",
      "stdout": "/var/log/backup.log"
    }
  ]
}
```

6.6.3 Example 3: Multiple Tasks

```
{
  "version": "1.0",
  "notify_command": "mail -s '[Bansuri]' admin@example.com",
  "scripts": [
    {
      "name": "quick-task",
      "command": "echo 'Quick'",
      "timer": "60"
    },
    {
      "name": "medium-task",
      "command": "sh /opt/process.sh",
      "timer": "300",
      "timeout": "2m",
      "on-fail": "restart",
      "times": 2
    },
    {
      "name": "critical-task",
      "command": "python /opt/critical.py",
      "timer": "3600",
      "timeout": "30m",
      "on-fail": "restart",
      "times": 3,
      "notify": "mail",
      "stdout": "/var/log/critical.log"
    }
  ]
}
```

6.7 Exit Codes Reference

Standard Exit Codes:

Code	Meaning
0	Success
1	General error
2	Misuse of shell command
126	Command invoked cannot execute
127	Command not found
128+N	Fatal signal N
130	Terminated (SIGINT/Ctrl+C)
137	Killed (SIGKILL)
143	Terminated (SIGTERM)

6.8 Testing Patterns

Test script exit code:

```
/path/to/script.sh
echo "Exit code: $?"
```

Test with different arguments:

```
/path/to/script.sh arg1 arg2
echo $?
```

Test with timeout:

```
timeout 30 /path/to/script.sh
echo $?
```

Test output redirection:

```
/path/to/script.sh > /tmp/out.log 2>&1
cat /tmp/out.log
```

6.9 Debugging Checklist

Task not running:

- [ ] Command is valid: `which command_name`
- [ ] Script is executable: `ls -la /path/to/script`
- [ ] Path is absolute in config
- [ ] Working directory exists: `ls -la /path/to/workdir`
- [ ] Timer is set (if periodic task)

Task failing repeatedly:

- [ ] Test command manually
- [ ] Check exit code: `/path/to/script.sh; echo $?`
- [ ] Check permissions: `ls -la /path/to/script`
- [ ] Check output: `tail -f /var/log/task.log`
- [ ] Check dependencies exist

Timeout issues:

- [ ] Measure actual execution: `time /path/to/script.sh`
- [ ] Increase timeout value
- [ ] Check system resources: `free -h`, `df -h`
- [ ] Check for blocking operations

Notifications not working:

- [ ] `notify` is set to `"mail"`
- [ ] `notify_command` is configured

- [ ] Test command: `echo "test" | mail -s "test" email@example.com`
- [ ] Check network connectivity
- [ ] Check Bansuri logs

6.10 Performance Tuning Checklist

Memory usage high:

- [ ] Redirect task output to files
- [ ] Reduce number of concurrent tasks
- [ ] Monitor: `ps aux | grep python`

CPU usage high:

- [ ] Increase timer intervals
- [ ] Check for infinite loops
- [ ] Profile: `python -m cProfile`

Tasks running slow:

- [ ] Check system load: `uptime`
- [ ] Monitor I/O: `iotop`
- [ ] Check network: `netstat -s`
- [ ] Profile execution: `time` command

6.11 Common Commands Cheat Sheet

```
# View configuration
cat scripts.json | python -m json.tool

# Validate configuration
python -c "from bansuri.base.config_manager import BansuriConfig; BansuriConfig.load_
↪from_file('scripts.json')"

# Find Bansuri process
pgrep -f "python -m bansuri"

# Kill Bansuri
pkill -f "python -m bansuri"

# Check system resources
free -h && df -h && uptime

# Monitor process
watch -n 1 'ps aux | grep bansuri'

# View recent logs
tail -50 /var/log/bansuri.log

# Search logs
```

(continues on next page)

(continued from previous page)

```
grep "ERROR" /var/log/bansuri.log

# Count task executions
grep -c "started\|completed" /var/log/bansuri.log
```

6.11.1 Useful Aliases

Add to `.bashrc`:

```
alias bansuri-status='systemctl status bansuri'
alias bansuri-logs='journalctl -u bansuri -f'
alias bansuri-restart='systemctl restart bansuri'
alias bansuri-config='cat ~/bansuri/scripts.json | python -m json.tool'
alias bansuri-validate='python -c "from bansuri.base.config_manager import BansuriConfig;
↪ BansuriConfig.load_from_file(\"scripts.json\"); print(\"✓ Valid\")"'
```

CHAPTER

SEVEN

ADVANCED CONFIGURATION

This guide covers advanced configuration features and patterns for Bansuri.

7.1 Timeout Management

7.1.1 Understanding Timeouts

The `timeout` parameter controls the maximum execution time for a task. If the task exceeds this time, the process is terminated.

7.1.2 Timeout Format

Timeouts can be specified as:

- **Seconds**: `"300"` or `300` (5 minutes)
- **Human-readable format**: - Seconds: `"30s"` → 30 seconds - Minutes: `"5m"` → 5 minutes (300 seconds) - Hours: `"2h"` → 2 hours (7200 seconds)

7.1.3 Examples

```
{
  "scripts": [
    {
      "name": "quick-task",
      "command": "echo 'Hello'",
      "timeout": "10s"
    },
    {
      "name": "medium-task",
      "command": "/usr/bin/backup.sh",
      "timeout": "30m"
    },
    {
      "name": "long-task",
      "command": "/opt/process/heavy-computation.py",
      "timeout": "1h"
    },
    {
      "name": "very-long-task",
      "command": "mysqldump production | gzip",
      "timeout": "3600"
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

7.2 Failure Control

7.2.1 Restart Policies

The `on-fail` and `times` parameters control task retry behavior:

- **on-fail: "stop"** (default): Stop the task immediately on failure
- **on-fail: "restart"**: Restart the task up to `times` attempts

The `success-codes` parameter specifies which exit codes are considered success (default: `[0]`).

7.2.2 Example Configurations

Single attempt, stop on failure:

```
{
  "name": "critical-task",
  "command": "deploy.sh",
  "on-fail": "stop",
  "times": 1
}
```

Three retry attempts:

```
{
  "name": "resilient-task",
  "command": "api-call.sh",
  "on-fail": "restart",
  "times": 3
}
```

Custom success codes:

```
{
  "name": "task-with-warnings",
  "command": "./checker.sh",
  "success-codes": [0, 2],
  "on-fail": "restart",
  "times": 2
}
```

7.3 Output Redirection

7.3.1 Log File Management

Control where task output goes with `stdout` and `stderr`:

- **stderr: "combined"** (default): Merge stderr into stdout

- **stderr: "/path/to/file"**: Redirect to a specific file
- **stdout: "/path/to/file"**: Redirect stdout separately

7.3.2 Examples

Combined output to single file:

```
{
  "name": "task-with-logs",
  "command": "backup.sh",
  "stdout": "/var/log/backup.log",
  "stderr": "combined"
}
```

Separate log files:

```
{
  "name": "task-separate-logs",
  "command": "process.py",
  "stdout": "/var/log/process.out",
  "stderr": "/var/log/process.err"
}
```

Log rotation pattern:

```
{
  "name": "task-with-rotation",
  "command": "data-sync.sh",
  "stdout": "/var/log/sync/$(date +%Y-%m-%d).log",
  "stderr": "combined"
}
```

7.4 Working Directory

The `working-directory` parameter sets the directory where the command is executed:

```
{
  "name": "repo-task",
  "command": "git pull && npm test",
  "working-directory": "/home/deploy/myapp"
}
```

This is useful for:

- Scripts that use relative paths
- Repository operations
- Docker container workloads

7.5 Task Descriptions

Add `description` field for documentation:

```
{
  "name": "backup-daily",
  "command": "pg_dump production > backup.sql",
  "timer": "86400",
  "description": "Daily PostgreSQL backup at midnight UTC"
}
```

7.6 Combining Features

Here's a comprehensive example using multiple advanced features:

```
{
  "version": "1.0",
  "notify_command": "mail -s '[Bansuri] Alert' ops@company.com",
  "scripts": [
    {
      "name": "critical-api-health-check",
      "command": "/opt/healthcheck/api-check.py",
      "timer": "60",
      "timeout": "30s",
      "working-directory": "/opt/healthcheck",
      "on-fail": "restart",
      "times": 3,
      "notify": "mail",
      "stdout": "/var/log/healthcheck.log",
      "stderr": "combined",
      "success-codes": [0, 1],
      "description": "API health check every minute with retries"
    },
    {
      "name": "nightly-database-backup",
      "command": "/usr/local/bin/backup-db.sh",
      "timer": "86400",
      "timeout": "2h",
      "working-directory": "/backups",
      "on-fail": "restart",
      "times": 2,
      "notify": "mail",
      "stdout": "/var/log/backup-$(date +%Y-%m-%d).log",
      "stderr": "combined",
      "description": "Nightly full database backup with retry"
    }
  ]
}
```

7.7 Performance Tuning

7.7.1 Task Concurrency

Each task runs in its own thread. The system can handle multiple concurrent tasks:

- **I/O-bound tasks**: Run many concurrently (network calls, file operations)
- **CPU-bound tasks**: Consider limiting concurrency and using appropriate timeouts
- **Resource-intensive tasks**: Monitor system resources

7.7.2 Timer-based Execution

The `timer` parameter defines fixed-interval execution:

```
{
  "name": "frequent-task",
  "command": "check.sh",
  "timer": "5",
  "description": "Runs every 5 seconds"
}
```

```
{
  "name": "hourly-task",
  "command": "hourly-job.sh",
  "timer": "3600",
  "description": "Runs every hour"
}
```

7.8 Error Handling Patterns

Pattern 1: Fail-fast

For critical tasks that should stop immediately:

```
{
  "name": "deploy",
  "command": "deploy.sh",
  "on-fail": "stop",
  "times": 1
}
```

Pattern 2: Resilient with retries

For tasks that might have transient failures:

```
{
  "name": "api-call",
  "command": "api-client.py",
  "on-fail": "restart",
  "times": 5,
  "timeout": "30s"
}
```

Pattern 3: Partial success acceptance

For tasks where certain exit codes are acceptable:

```
{
  "name": "data-check",
  "command": "validator.py",
  "success-codes": [0, 2],
  "notify": "mail"
}
```

Pattern 4: Monitored execution with notification

For important tasks that need monitoring:

```
{
  "name": "critical-job",
  "command": "critical.sh",
  "timer": "3600",
  "timeout": "30m",
  "on-fail": "restart",
  "times": 3,
  "notify": "mail",
  "stdout": "/var/log/critical.log",
  "stderr": "combined"
}
```

7.9 Common Mistakes to Avoid

1. **Setting timeout too short**: Leave enough time for normal execution
2. **Ignoring failure modes**: Test on-fail behavior
3. **Not managing logs**: Use log rotation or separate log directories
4. **Forgetting working-directory**: Relative paths may fail
5. **Ignoring resource limits**: Monitor CPU and memory usage

7.10 Debugging Configuration Issues

Enable debug output:

```
export BANSURI_LOG_LEVEL=DEBUG
bansuri --config scripts.json
```

Check configuration syntax:

```
python -c "
from bansuri.base.config_manager import BansuriConfig
config = BansuriConfig.load_from_file('scripts.json')
for script in config.scripts:
    print(f'{script.name}: OK')
"
```

Test command execution:

```
bash -c "cd /working/dir && /path/to/command"
```

CHAPTER

EIGHT

USAGE

8.1 Basic Usage

To start the orchestration system from the command line:

```
bansuri
```

This will load `scripts.json` from the current working directory and start monitoring tasks according to the configuration.

To specify a different configuration file:

```
bansuri --config /path/to/scripts.json
```

8.1.1 Programmatic Usage

You can also use Bansuri as a library:

```python
from bansuri.master import Orchestrator
from bansuri.base.config_manager import BansuriConfig

# Load configuration
config = BansuriConfig.load_from_file("scripts.json")

# Create and start orchestrator
orchestrator = Orchestrator(config_file="scripts.json")
orchestrator.start()
```

8.2 Configuration

Bansuri is configured primarily through the `scripts.json` file.

For a detailed guide on how to add new scripts or modify existing parameters, please refer to the *Configuration* guide.

CHAPTER

NINE

NOTIFICATIONS

Bansuri includes a built-in notification system to alert you when tasks fail. This allows you to respond quickly to issues in your orchestrated tasks.

9.1 Overview

The notification system is triggered when a task fails (returns a non-zero exit code) and the `notify` configuration option is set to `"mail"`. Notifications are sent using a command-based system, allowing integration with any notification service (email, Slack, webhooks, etc.).

9.2 Configuration

To enable notifications, you need to:

1. Set the `notify` field in your task configuration to `"mail"`
2. Configure a global `notify_command` in your `scripts.json`

9.2.1 Example Configuration

```
{
  "version": "1.0",
  "notify_command": "mail -s '[Bansuri] Task Failed' your-email@example.com",
  "scripts": [
    {
      "name": "critical-backup",
      "command": "/usr/local/bin/backup.sh",
      "timer": "3600",
      "notify": "mail",
      "on-fail": "restart",
      "times": 3
    }
  ]
}
```

9.2.2 Configuration Parameters

- **notify_command** (global): Shell command to execute for sending notifications. The command receives the failure message as input.

  Examples:

  - Email: `"mail -s '[Bansuri] Task Failed' admin@example.com"`
  - Slack: `"curl -X POST -d @- https://hooks.slack.com/services/YOUR/WEBHOOK/URL"`
  - Webhook: `"curl -X POST -H 'Content-Type: application/json' -d @- https://your-api.example.com/notify"`

- **notify** (per-task): Set to `"mail"` to enable notifications for this task, or `"none"` to disable.

9.3 Failure Information

When a task fails, Bansuri sends the following information in the notification:

- **Task Name**: Unique identifier of the failed task
- **Command**: The command that was executed
- **Working Directory**: The directory where the command ran
- **Return Code**: Exit code of the failed process
- **Attempt**: Which attempt this was (e.g., "2/3")
- **Timestamp**: When the failure occurred
- **Description**: Task description (if provided)
- **Stdout**: Standard output captured from the failed task
- **Stderr**: Standard error output captured from the failed task

9.3.1 Notification Message Format

The notification is formatted as follows:

```
=== [Bansuri] Task Failure ===

Task 'task-name' has failed.

--- Task Details ---
Name:              task-name
Command:           /path/to/command
Working Directory: /working/dir
Return Code:       1
Attempt:           2/3
Timestamp:         2026-01-19 14:30:45
Description:       Optional task description

--- Output ---
(stdout from task)

--- Error ---
(stderr from task)

---
This is an automated message from Bansuri Orchestrator.
```

9.4 API Reference

9.4.1 Notifier Base Class

class `bansuri.alerts.notifier.`**`Notifier`**

Bases: `ABC`

Base class for notification handlers.

abstractmethod notify(*failure_info:* FailureInfo) → bool

Abstract method that handles on failure notifications

9.4.2 FailureInfo Data Class

class `bansuri.alerts.notifier.`**`FailureInfo`**(*task_name: str*, *command: str*, *working_directory: str* | *None*, *return_code: int*, *attempt: int*, *max_attempts: int*, *timestamp: datetime.datetime*, *description: str*, *stdout: str*, *stderr: str*)

attempt: int

command: str

description: str

max_attempts: int

return_code: int

stderr: str

stdout: str

task_name: str

timestamp: datetime

working_directory: str | None

9.4.3 CommandNotifier Implementation

class `bansuri.alerts.cmd_notifier.`**`CommandNotifier`**(*notify_command: str*, *timeout: int = 30*)

Bases: *`Notifier`*

Notify via shell command execution.

notify(*failure_info:* FailureInfo) → bool

Abstract method that handles on failure notifications

9.5 Examples

9.5.1 Email Notifications

Send notifications via email using the `mail` command:

```
{
  "notify_command": "mail -s '[Bansuri Alert] Task Failed' ops-team@company.com",
  "scripts": [
    {
      "name": "database-backup",
      "command": "pg_dump production > /backups/db.sql",
      "timer": "86400",
      "notify": "mail",
      "on-fail": "restart",
      "times": 2
    }
  ]
}
```

9.5.2 Slack Notifications

Send notifications to a Slack channel:

```
{
  "notify_command": "curl -X POST -d @- https://hooks.slack.com/services/YOUR/WEBHOOK/URL
↪",
  "scripts": [
    {
      "name": "critical-service",
      "command": "systemctl restart myservice",
      "timer": "300",
      "notify": "mail",
      "on-fail": "stop"
    }
  ]
}
```

9.5.3 Custom Webhook

Send notifications to a custom API endpoint:

```
{
  "notify_command": "curl -X POST -H 'Content-Type: application/json' -d @- https://
↪monitoring.company.com/api/alerts",
  "scripts": [
    {
      "name": "health-check",
      "command": "/opt/health-check.py",
      "timer": "600",
      "notify": "mail",
      "on-fail": "restart",
      "times": 3
    }
  ]
}
```

9.6 Extending the Notification System

You can create custom notifier implementations by subclassing the `Notifier` base class:

```
from bansuri.alerts.notifier import Notifier, FailureInfo

class SlackNotifier(Notifier):
    def __init__(self, webhook_url: str):
        self.webhook_url = webhook_url

    def notify(self, failure_info: FailureInfo) -> bool:
        # Your implementation to send to Slack
        message = f"Task '{failure_info.task_name}' failed with code {failure_info.
↪return_code}"
        # ... send to Slack ...
        return True
```

Then modify the `TaskRunner._create_notifier()` method to use your custom notifier.

9.7 Troubleshooting

9.7.1 Notifications not being sent

Issue: Tasks are failing but no notifications are received.

Solutions:

1. Verify the `notify` field is set to `"mail"` in the task configuration
2. Check that `notify_command` is defined at the global level in `scripts.json`
3. Test the notify command manually:

   ```
   echo "Test message" | mail -s "Test" your-email@example.com
   ```

4. Check Bansuri logs for error messages related to notification sending
4. Ensure the user running Bansuri has permissions to execute the notify command

9.7.2 Notification timeout

Issue: Notifications are causing tasks to hang or slow down.

Solution: The `CommandNotifier` has a default 30-second timeout. To increase it, modify the timeout in `TaskRunner._create_notifier()`:

```
return CommandNotifier(notify_cmd, timeout=60)  # 60 seconds
```

9.8 Integration Notes

- Notifications are sent **asynchronously** within the task execution thread, so they won't block task execution.
- If a notification fails, it's logged but doesn't affect the task execution flow.
- The notification message is formatted with escaped newlines (\\n) for compatibility with shell commands.

CHAPTER

TEN

CREATING CUSTOM TASKS

Bansuri provides a powerful extension system that allows you to create custom task implementations beyond simple shell commands. This is perfect for complex business logic, Python scripts with special requirements, or integrations with other systems.

10.1 Overview

The base class `AbstractTask` defines the interface for all executable tasks in Bansuri. By inheriting from this class and implementing the required methods, you can create sophisticated task handlers.

10.1.1 Abstract Base Class

class `bansuri.base.task_base.`**`AbstractTask`**(*config: TaskConfig*)

Bases: `ABC`

Abstract base class for tasks. Uses a modular TaskConfig for properties and as execution interface.

abstractmethod run() → int

Execute the task. :returns: The exit code of the command. :rtype: int

abstractmethod stop() → None

Terminate the task if it is running.

10.2 Basic Example: Logging Task

Here's a simple custom task that logs messages to a file:

```python
from bansuri.base.task_base import AbstractTask
from bansuri.base.config.task_config import TaskConfig
from datetime import datetime

class LoggingTask(AbstractTask):
    """A simple task that logs to a file."""

    def __init__(self, config: TaskConfig):
        super().__init__(config)
        self.running = False

    def run(self) -> int:
        """Execute the logging task."""
        self.running = True
```

(continues on next page)

(continued from previous page)

```
        try:
            log_file = self.config.logging.stdout_path or "/tmp/task.log"
            message = f"[{datetime.now()}] Task executed: {self.config.identification.
↪name}"

            with open(log_file, "a") as f:
                f.write(message + "\n")

            return 0  # Success
        except Exception as e:
            print(f"Error: {e}")
            return 1  # Failure
        finally:
            self.running = False

    def stop(self) -> None:
        """Stop the task if running."""
        self.running = False
```

10.3 Advanced Example: Database Backup Task

A more realistic example that backs up a PostgreSQL database:

```
import subprocess
import os
from datetime import datetime
from bansuri.base.task_base import AbstractTask
from bansuri.base.config.task_config import TaskConfig

class PostgresBackupTask(AbstractTask):
    """Backs up a PostgreSQL database with compression."""

    def __init__(self, config: TaskConfig):
        super().__init__(config)
        self.process = None

    def run(self) -> int:
        """Execute the backup."""
        try:
            # Get configuration from task config
            db_name = os.environ.get("DB_NAME", "production")
            backup_dir = self.config.logging.stdout_path or "/backups"
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            backup_file = f"{backup_dir}/{db_name}_{timestamp}.sql.gz"

            # Create backup command
            cmd = f"pg_dump {db_name} | gzip > {backup_file}"

            # Execute with timeout
            timeout = self._parse_timeout()
            self.process = subprocess.Popen(
```

(continues on next page)

(continued from previous page)

```
                cmd,
                shell=True,
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE,
                cwd=self.config.identification.working_directory
            )

            try:
                stdout, stderr = self.process.communicate(timeout=timeout)
                return self.process.returncode
            except subprocess.TimeoutExpired:
                self.process.kill()
                return -1  # Timeout

        except Exception as e:
            print(f"Backup failed: {e}")
            return 1

    def stop(self) -> None:
        """Terminate the backup process."""
        if self.process and self.process.poll() is None:
            self.process.terminate()
            try:
                self.process.wait(timeout=5)
            except subprocess.TimeoutExpired:
                self.process.kill()

    def _parse_timeout(self) -> int:
        """Parse timeout from config."""
        timeout_str = self.config.scheduling.timeout
        if not timeout_str:
            return 3600  # Default 1 hour

        if isinstance(timeout_str, int):
            return timeout_str

        # Parse formats like "30s", "5m", "1h"
        timeout_str = str(timeout_str).lower()
        if timeout_str.endswith("s"):
            return int(timeout_str[:-1])
        elif timeout_str.endswith("m"):
            return int(timeout_str[:-1]) * 60
        elif timeout_str.endswith("h"):
            return int(timeout_str[:-1]) * 3600
        else:
            return int(timeout_str)
```

10.4 Advanced Example: Health Check Task

A task that performs system health checks and reports metrics:

```python
import json
import subprocess
from datetime import datetime
from bansuri.base.task_base import AbstractTask
from bansuri.base.config.task_config import TaskConfig

class HealthCheckTask(AbstractTask):
    """Performs system health checks and sends metrics."""

    def __init__(self, config: TaskConfig):
        super().__init__(config)
        self.metrics = {}

    def run(self) -> int:
        """Execute health checks."""
        try:
            # CPU usage
            cpu_result = subprocess.run(
                "grep 'cpu ' /proc/stat",
                shell=True,
                capture_output=True,
                text=True
            )

            # Memory usage
            mem_result = subprocess.run(
                "free -b | grep Mem",
                shell=True,
                capture_output=True,
                text=True
            )

            # Disk usage
            disk_result = subprocess.run(
                "df -B1 /",
                shell=True,
                capture_output=True,
                text=True
            )

            self.metrics = {
                "timestamp": datetime.now().isoformat(),
                "cpu": cpu_result.stdout.strip(),
                "memory": mem_result.stdout.strip(),
                "disk": disk_result.stdout.strip()
            }

            # Send metrics to monitoring system
            return self._send_metrics()
```

(continues on next page)

(continued from previous page)

```
        except Exception as e:
            print(f"Health check failed: {e}")
            return 1

    def stop(self) -> None:
        """Clean up health check task."""
        pass

    def _send_metrics(self) -> int:
        """Send metrics to a monitoring backend."""
        # This would integrate with your monitoring system
        # (Prometheus, Grafana, InfluxDB, etc.)
        print(json.dumps(self.metrics))
        return 0
```

10.5 Configuration for Custom Tasks

When using custom tasks, you need to:

1. **Set ``no-interface`` to ``true``** in the task configuration
2. **Reference the module path** in the command field (Python import path)

10.5.1 Example Configuration

```
{
  "version": "1.0",
  "scripts": [
    {
      "name": "custom-logging-task",
      "command": "myapp.tasks:LoggingTask",
      "no-interface": true,
      "timer": "3600",
      "description": "Runs custom logging task every hour"
    },
    {
      "name": "postgres-backup",
      "command": "myapp.tasks:PostgresBackupTask",
      "no-interface": true,
      "timer": "86400",
      "timeout": "2h",
      "on-fail": "restart",
      "times": 2,
      "description": "Daily PostgreSQL backup with retry"
    }
  ]
}
```

10.6 Best Practices

1. **Exception Handling**: Always wrap your logic in try-except blocks
2. **Return Codes**: Use 0 for success, non-zero for failure
3. **Graceful Shutdown**: Implement proper cleanup in the `stop()` method
4. **Logging**: Use standard logging for debugging and error tracking
5. **Timeout Support**: Parse and respect the timeout configuration
6. **Resource Cleanup**: Close files, connections, and processes properly

10.6.1 Example with Logging

```
import logging
from bansuri.base.task_base import AbstractTask

class BestPracticeTask(AbstractTask):
    def __init__(self, config):
        super().__init__(config)
        self.logger = logging.getLogger(config.identification.name)

    def run(self) -> int:
        try:
            self.logger.info(f"Starting task: {self.config.identification.name}")
            # Do work
            self.logger.info("Task completed successfully")
            return 0
        except Exception as e:
            self.logger.error(f"Task failed: {e}", exc_info=True)
            return 1

    def stop(self) -> None:
        self.logger.info("Stopping task")
```

10.7 Testing Custom Tasks

Example test for a custom task:

```
import unittest
from myapp.tasks import LoggingTask
from bansuri.base.config.task_config import TaskConfig
from bansuri.base.config.proc_id_config import IdentificationConfig

class TestLoggingTask(unittest.TestCase):
    def setUp(self):
        self.config = TaskConfig(
            identification=IdentificationConfig(
                name="test-task",
                command="test"
            )
        )
```

(continues on next page)

(continued from previous page)

```
        self.task = LoggingTask(self.config)

    def test_run_success(self):
        result = self.task.run()
        self.assertEqual(result, 0)

    def test_stop(self):
        self.task.stop()  # Should not raise

if __name__ == "__main__":
    unittest.main()
```

10.8 Integration with Bansuri

Custom tasks are automatically detected and loaded by the `TaskRunner` when:

1. `no-interface` is set to `true`
2. The `command` field contains a valid Python module path and class name (format: `module.path:ClassName`)

The system will:

1. Import the module
2. Instantiate the class with the `TaskConfig`
3. Call `run()` in a thread
4. Call `stop()` when the orchestrator shuts down or the task is removed

See *Notifications* for how failures are handled and notifications are sent.

CHAPTER

ELEVEN

DEPLOYMENT GUIDE

This guide covers deployment strategies for production use of Bansuri.

11.1 Systemd Service Setup

11.1.1 Running as a Systemd Service

Create a systemd unit file at `/etc/systemd/system/bansuri.service`:

```
[Unit]
Description=Bansuri Task Orchestrator
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=bansuri
Group=bansuri
WorkingDirectory=/opt/bansuri
ExecStart=/opt/bansuri/venv/bin/python -m bansuri

# Restart policy
Restart=on-failure
RestartSec=10

# Resource limits
MemoryLimit=1G
CPUQuota=80%

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=bansuri

# Environment
Environment="BANSURI_ENV=production"
Environment="BANSURI_LOG_LEVEL=INFO"

[Install]
WantedBy=multi-user.target
```

Enable and start the service:

```
sudo systemctl daemon-reload
sudo systemctl enable bansuri
sudo systemctl start bansuri
sudo systemctl status bansuri
```

View logs:

```
journalctl -u bansuri -f
```

11.2 Docker Deployment

11.2.1 Dockerfile

```
FROM python:3.11-slim

# Create bansuri user
RUN useradd -m -u 1000 bansuri

# Install dependencies
RUN apt-get update && apt-get install -y \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Install Bansuri
RUN mkdir -p /opt/bansuri && chown bansuri:bansuri /opt/bansuri
WORKDIR /opt/bansuri

COPY --chown=bansuri:bansuri . .
RUN pip install --no-cache-dir -e .

# Create log directory
RUN mkdir -p /var/log/bansuri && chown bansuri:bansuri /var/log/bansuri

USER bansuri

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
    CMD python -c "import sys; sys.exit(0)" || exit 1

CMD ["python", "-m", "bansuri"]
```

11.2.2 Docker Compose

```
version: '3.8'

services:
  bansuri:
    build: .
    container_name: bansuri
    user: bansuri
```

(continues on next page)

(continued from previous page)

```
    volumes:
      - ./scripts.json:/opt/bansuri/scripts.json:ro
      - ./logs:/var/log/bansuri
      - /var/run/docker.sock:/var/run/docker.sock:ro
    environment:
      - BANSURI_ENV=production
      - BANSURI_LOG_LEVEL=INFO
    restart: unless-stopped
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

11.3 Kubernetes Deployment

11.3.1 ConfigMap for Configuration

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: bansuri-config
  namespace: default
data:
  scripts.json: |
    {
      "version": "1.0",
      "notify_command": "curl -X POST https://alerts.example.com/notify",
      "scripts": [
        {
          "name": "health-check",
          "command": "/usr/local/bin/health-check.sh",
          "timer": "60",
          "timeout": "30s"
        }
      ]
    }
```

11.3.2 Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bansuri
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bansuri
```

(continues on next page)

(continued from previous page)

```yaml
  template:
    metadata:
      labels:
        app: bansuri
    spec:
      containers:
      - name: bansuri
        image: myregistry.azurecr.io/bansuri:latest
        imagePullPolicy: Always
        resources:
          limits:
            memory: "1Gi"
            cpu: "500m"
          requests:
            memory: "512Mi"
            cpu: "250m"
        env:
        - name: BANSURI_ENV
          value: "production"
        - name: BANSURI_LOG_LEVEL
          value: "INFO"
        volumeMounts:
        - name: config
          mountPath: /opt/bansuri/scripts.json
          subPath: scripts.json
        - name: logs
          mountPath: /var/log/bansuri
        livenessProbe:
          exec:
            command:
            - python
            - -c
            - "import sys; sys.exit(0)"
          initialDelaySeconds: 30
          periodSeconds: 10
      volumes:
      - name: config
        configMap:
          name: bansuri-config
      - name: logs
        emptyDir: {}
      serviceAccountName: bansuri
```

11.3.3 ServiceAccount

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bansuri
  namespace: default
```

11.4 Production Checklist

Before deploying to production, verify:

Configuration

- All task commands are absolute paths
- Working directories are correctly specified
- Timeouts are realistic for your infrastructure
- Notification endpoints are configured and tested
- Log directories exist and are writable

Security

- Run as non-root user
- Use restrictive file permissions (600-755)
- Verify all scripts are from trusted sources
- Use secrets management for credentials
- Enable audit logging

Monitoring

- Set up log aggregation (ELK, Splunk, etc.)
- Configure alerting for task failures
- Monitor system resources (CPU, memory, disk)
- Set up health checks for Bansuri itself
- Create dashboards for task execution metrics

Resilience

- Test failure scenarios (network, timeout, out of memory)
- Verify restart policies
- Test graceful shutdown (SIGTERM handling)
- Ensure log rotation is working
- Validate recovery time objectives

Documentation

- Document all tasks and their purpose
- Document task dependencies
- Maintain runbooks for common issues
- Document escalation procedures
- Keep version history of configurations

11.5 Backup and Recovery

11.5.1 Configuration Backup

```
# Backup scripts.json with timestamp
cp scripts.json "scripts.json.backup.$(date +%Y%m%d_%H%M%S)"

# Keep last 30 days of backups
find . -name "scripts.json.backup.*" -mtime +30 -delete
```

11.5.2 Log Backup

```
# Archive logs older than 7 days
find /var/log/bansuri -name "*.log" -mtime +7 -exec gzip {} \;
find /var/log/bansuri -name "*.log.gz" -mtime +30 -delete
```

11.5.3 Recovery Procedure

1. Stop Bansuri: `systemctl stop bansuri`
2. Restore configuration: `cp scripts.json.backup.* scripts.json`
3. Verify configuration: ``python -c "from bansuri.base.config_manager import BansuriConfig; BansuriConfig.load_from_file('scripts.json')"`
4. Start Bansuri: `systemctl start bansuri`
5. Verify status: `systemctl status bansuri`

11.6 Scaling Considerations

11.6.1 Single Instance

- Suitable for < 100 concurrent tasks
- Sufficient for < 1000 events per minute
- Simple deployment and management

11.6.2 Multiple Instances

For larger deployments:

1. **Task Distribution**: Partition tasks across instances
2. **Configuration Sync**: Use centralized config storage (Git, S3, etc.)
3. **Log Aggregation**: Send logs to centralized location
4. **Health Monitoring**: Monitor all instances

Example multi-instance setup:

```
# Instance 1: High-priority critical tasks
bansuri --config scripts-critical.json

# Instance 2: Background/batch tasks
```

(continues on next page)

(continued from previous page)

```
bansuri --config scripts-batch.json

# Instance 3: Monitoring/health-check tasks
bansuri --config scripts-monitoring.json
```

11.7 Performance Tuning

11.7.1 Memory Management

Monitor memory usage:

```
# Monitor Bansuri process
watch -n 1 'ps aux | grep bansuri'
```

Set limits in systemd:

```
[Service]
MemoryLimit=2G
MemoryAccounting=yes
```

11.7.2 CPU Affinity

Bind Bansuri to specific CPU cores:

```
[Service]
CPUAffinity=0-3
```

For systemd systems with taskset:

```
taskset -cp 0-3 $(pgrep -f "python -m bansuri")
```

11.8 Troubleshooting Deployment Issues

11.8.1 Service won't start

```
# Check syntax
systemctl status bansuri
journalctl -u bansuri -n 50

# Verify permissions
ls -la /opt/bansuri
sudo -u bansuri python -m bansuri --help
```

11.8.2 High memory usage

```
# Check for zombie processes
ps aux | grep defunct

# Monitor task output
tail -f /var/log/bansuri/task.log
```

(continues on next page)

(continued from previous page)

```
# Reduce concurrency
# Modify scripts.json to reduce number of concurrent tasks
```

11.8.3 Tasks not running

```
# Verify configuration
python -m bansuri --validate-config

# Check file permissions on log directories
ls -la /var/log/bansuri

# Check if user has execute permissions
sudo -u bansuri which /path/to/command
```

CHAPTER

TWELVE

TROUBLESHOOTING

Common problems and solutions.

12.1 Task Issues

12.1.1 Task Won't Start

Symptom: Task stays `PENDING` or immediately fails

Checklist:

1. Does `scripts.json` have valid JSON syntax?

   ```
   python -m json.tool scripts.json
   ```

2. Is the `command` executable?

   ```
   # Test the command directly
   bash -x "your-command"
   ```

3. Does the file exist and have permissions?

   ```
   ls -l /path/to/script.sh
   chmod +x /path/to/script.sh
   ```

4. Are you using an absolute path?

   Good:

   ```
   "command": "/usr/local/bin/backup.sh"
   ```

   Bad:

   ```
   "command": "backup.sh"
   ```

—

12.1.2 Task Runs But Fails

Symptom: Task exits with non-zero code

Checklist:

1. Check exit code in logs

```
[task] Process finished with code 127   # Command not found
[task] Process finished with code 1     # General error
[task] Process finished with code 126   # Permission denied
```

2. Run command manually to see error:

```
bash -x /path/to/script.sh
```

3. Check for missing environment variables:

```
"working-directory": "/app",
"command": "python main.py"
```

4. Consider adding to `success-codes` if non-zero is acceptable:

```
"success-codes": [0, 1]
```

—

12.1.3 Task Runs Forever

Symptom: Task completes but `RUNNING` status persists

Fix: Set a `timeout`

```
{
  "name": "hanging-task",
  "command": "python task.py",
  "timeout": "30s"
}
```

—

12.1.4 High Memory Usage

Symptom: Bansuri process grows in memory

Causes:

- Task produces huge output
- Many tasks logging to memory

Solutions:

1. Redirect output to files:

```
"stdout": "/var/log/task.log",
"stderr": "combined"
```

2. Or in the command itself:

```
"command": "python task.py > /tmp/output.log 2>&1"
```

3. Limit number of tasks

—

12.2 Configuration Issues

12.2.1 "croniter not found" Error

Symptom: When using `schedule-cron`

```
ERROR: 'croniter' library is missing
```

Fix:

```
pip install croniter
```

—

12.2.2 Notification Not Sending

Symptom: Task fails but no email arrives

Checklist:

1. Is `notify` set to `"mail"`?

   ```
   "notify": "mail"
   ```

2. Is `notify_command` configured?

   ```
   "notify_command": "mail -s 'Alert: {task}' admin@example.com"
   ```

3. Is `mail` command available?

   ```
   which mail
   # If missing:
   sudo apt-get install mailutils   # Ubuntu
   sudo yum install mailx            # CentOS
   ```

4. Test mail manually:

   ```
   echo "test" | mail -s "Test" admin@example.com
   ```

—

12.2.3 Invalid Configuration

Symptom: Tasks ignored, no error messages

Check:

1. JSON syntax:

   ```
   python -m json.tool scripts.json
   ```

2. Required fields:

   ```
   {
     "name": "...",        // REQUIRED
     "command": "...",     // REQUIRED
     "version": "1.0"      // Recommended
   }
   ```

3. Validate with Python:

   ```
   from bansuri.base.config_manager import BansuriConfig
   config = BansuriConfig.load_from_file('scripts.json')
   ```

—

12.2.4 Cron Not Working

Symptom: `schedule-cron` doesn't trigger

Check cron expression:

```
from croniter import croniter
from datetime import datetime

cron = "0 2 * * *"  # Your expression
if croniter.is_valid(cron):
    print("Valid")
    c = croniter(cron)
    print(f"Next run: {c.get_next(datetime)}")
else:
    print("Invalid cron expression")
```

Common cron patterns:

```
"0 2 * * *"      → Daily at 2 AM
"*/5 * * * *"    → Every 5 minutes
"0 */6 * * *"    → Every 6 hours
"0 0 1 * *"      → Monthly on 1st
```

—

12.3 Restart / Recovery

12.3.1 Restart Bansuri

```
# If running systemd
sudo systemctl restart bansuri

# If running manually
Ctrl+C  # Stop
bansuri # Restart
```

—

12.3.2 Force Stop Hung Task

```
# Find process
ps aux | grep bansuri

# Send signal
kill -15 <PID>       # Graceful
kill -9 <PID>        # Force
```

—

12.3.3 Reset All Tasks

Stop Bansuri and remove any state files:

```
# Stop Bansuri
bansuri stop

# Optional: Clean logs
rm /var/log/bansuri.log*

# Restart
bansuri
```

—

12.4 Getting Help

12.4.1 Enable Debug Logging

```
# Check current logs (follow mode)
tail -f /var/log/bansuri.log | grep ERROR

# Look for specific task
grep "task-name" /var/log/bansuri.log
```

12.4.2 Check Bansuri Status

```
# Show running processes
ps aux | grep bansuri

# Check ports (if applicable)
netstat -tlnp | grep bansuri

# Show config being used
cat scripts.json | python -m json.tool
```

12.4.3 Need More Help?

1. Check ../NOT_IMPLEMENTED.md for known limitations
2. Review *Configuration* for correct parameter format
3. See *Reference* for examples
4. Visit GitHub issues: https://github.com/aziabatz/bansuri/issues

CHAPTER

THIRTEEN

CONTRIBUTING TO BANSURI

Thank you for your interest in contributing! We welcome pull requests from developers of all skill levels.

13.1 Setting up the Development Environment

1. **Clone the repository**

   ```
   git clone https://github.com/aziabatz/bansuri.git
   cd bansuri
   ```

2. **Create a virtual environment**

   ```
   python -m venv venv
   source venv/bin/activate
   ```

3. **Install the package in editable mode**

   ```
   pip install -e .
   ```

13.2 Project Structure

- `bansuri/`: Main package source code. * `master.py`: Orchestrator - main orchestration logic and task coordination. * `task_runner.py`: TaskRunner - task execution and subprocess management. * `base/`: Base classes and configuration management.
  - `task_base.py`: AbstractTask - base class for custom task implementations.
  - `config_manager.py`: BansuriConfig and ScriptConfig - configuration data structures.
  - `config/`: Configuration component classes (Identification, Scheduling, Failure Control, etc.).
  - `misc/`: Miscellaneous utilities (header, help).
  - `alerts/`: Notification system. * `notifier.py`: Base notifier interface. * `cmd_notifier.py`: Command-based notifications.
- `tests/`: Unit and integration tests.
- `doc/`: Sphinx documentation source.
- `examples/`: Example scripts and configurations.

13.3 Running Tests

We use `pytest` for testing. Ensure all tests pass before submitting a PR:

```
pytest test_scripts.py
pytest test_notify.py
```

13.4 Documentation

If you modify the code, please update the docstrings and regenerate the documentation to ensure consistency.

13.4.1 Building Documentation

To build the Sphinx documentation:

```
cd doc
make html
```

The generated documentation will be in `doc/_build/html/`.

CHAPTER

FOURTEEN

API REFERENCE

14.1 Core Components

14.1.1 Orchestrator

`class bansuri.master.Orchestrator`(*config_file='scripts.json'*, *check_interval=30*)

Bases: `object`

`run`()

`signal_handler`(*signum*, *frame*)

POSIX signal handling

Parameters

- **`signum`** – Signal identifier
- **`frame`** – Unused

`stop_all`()

`sync_tasks`()

Synchronize tasks from config file.

14.1.2 TaskRunner

`class bansuri.task_runner.TaskRunner`(*config:* ScriptConfig, *bansuri_config:* BansuriConfig)

Bases: `object`

This class manages the lifecycle of a single task. It handles process execution, log redirection, and other policies.

`log`(*message: str*)

Fallback formatted log output

`start`()

Starts the control thread if it is stopped

`stop`()

Sends the termination signal

14.1.3 Configuration

class `bansuri.base.config_manager.`**`BansuriConfig`**(*version: str*, *scripts: List[*ScriptConfig*]*, *notify_command: str | None = None*)

Bases: `object`

Represents the current loaded definitions for Bansuri

classmethod load_from_file(*file_path: str*) → *BansuriConfig*

notify_command: str | None = None

scripts: List[*ScriptConfig*]

version: str

class `bansuri.base.config_manager.`**`ScriptConfig`**(*name: str*, *command: str*, *user: str | None = None*, *working_directory: str | None = None*, *no_interface: bool = False*, *schedule_cron: str | None = None*, *timer: str | None = None*, *timeout: str | None = None*, *times: int = 1*, *on_fail: str = 'stop'*, *depends_on: List[str] = <factory>*, *success_codes: List[int] = <factory>*, *environment_file: str | None = None*, *priority: int = 0*, *stdout: str | None = None*, *stderr: str | None = 'combined'*, *notify: str = 'none'*, *description: str = ''*)

Bases: `object`

Represents the configuration of a task

command: str

depends_on: List[str]

description: str = ''

environment_file: str | None = None

property is_smart_script: bool

name: str

no_interface: bool = False

notify: str = 'none'

on_fail: str = 'stop'

priority: int = 0

schedule_cron: str | None = None

stderr: str | None = 'combined'

stdout: str | None = None

success_codes: List[int]

timeout: str | None = None

timer: str | None = None

times: int = 1

user: str | None = None

validate()

Applies differentiated validation rules depending on the script type.

working_directory: str | None = None

14.2 Base Classes for Extensions

14.2.1 AbstractTask

`class bansuri.base.task_base.`**`AbstractTask`**`(`*config: TaskConfig*`)`

Bases: `ABC`

Abstract base class for tasks. Uses a modular TaskConfig for properties and as execution interface.

abstractmethod run() → int

Execute the task. :returns: The exit code of the command. :rtype: int

abstractmethod stop() → None

Terminate the task if it is running.

14.2.2 Notification System

`class bansuri.alerts.notifier.`**`Notifier`**

Bases: `ABC`

Base class for notification handlers.

abstractmethod notify(*failure_info:* FailureInfo) → bool

Abstract method that handles on failure notifications

`class bansuri.alerts.cmd_notifier.`**`CommandNotifier`**`(`*notify_command: str*, *timeout: int = 30*`)`

Bases: *`Notifier`*

Notify via shell command execution.

notify(*failure_info:* FailureInfo) → bool

Abstract method that handles on failure notifications

Note

Check NOT_IMPLEMENTED.md for feature status and workarounds.

CHAPTER FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search