MDSplus Redis Based Distributed Dispatcher for ITER Neutral Beam Test Facility

Nuno Cruz[a,b], Gabriele Manduchi[b], Cesare Taliercio[b], Joshua Stillerman[c]

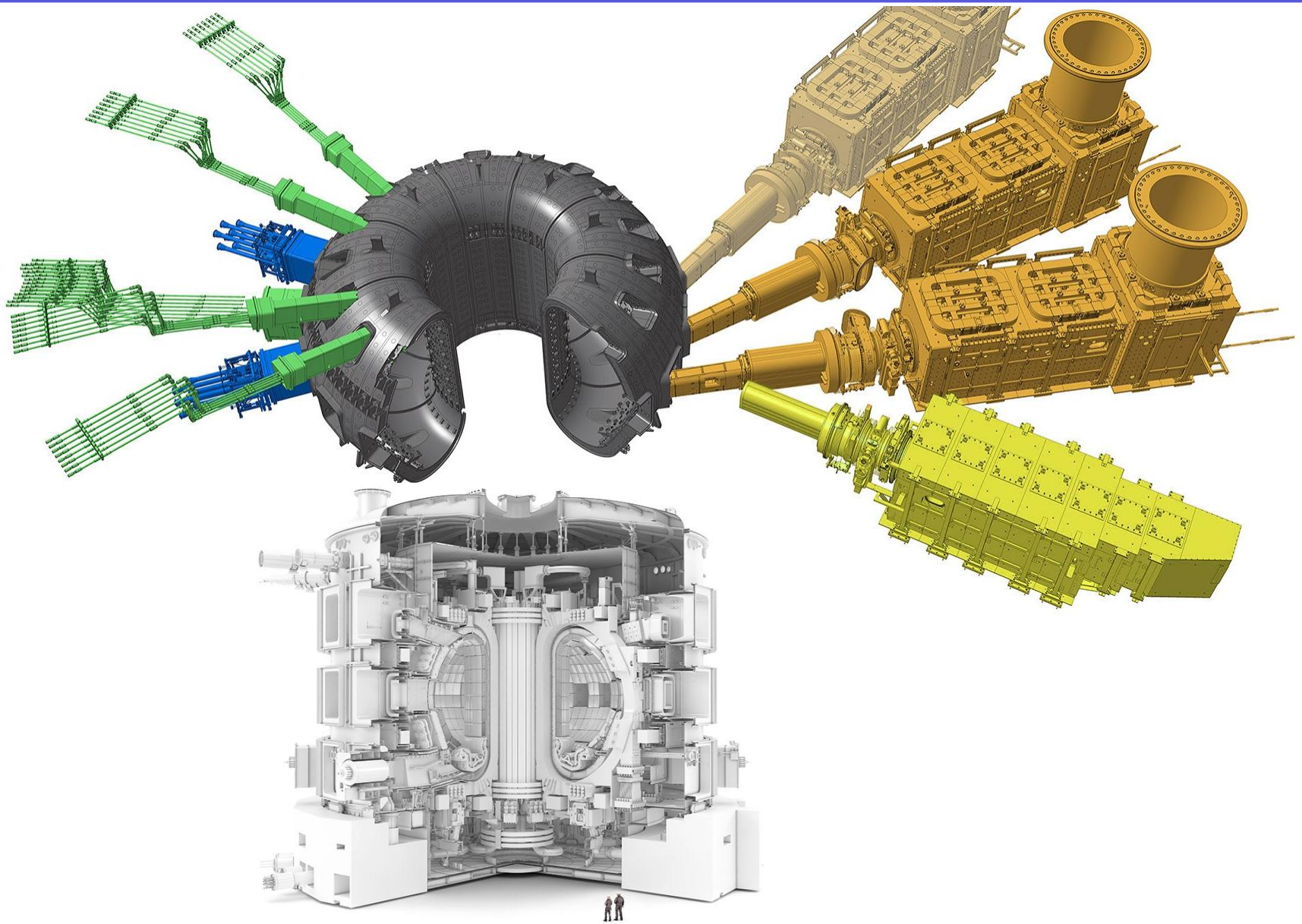
[a] Instituto de Plasmas e Fusão Nuclear, Instituto Superior Técnico, Universidade de Lisboa, 1049-001, Lisboa, Portugal

[b] Consorzio RFX, Corso Stati Uniti, 4, 35127 Padova Italy

[c] MIT Plasma Science and Fusion Center, Cambridge, MA, United States of America

25th IEEE NPSS Real Time Conference
25–29 May 2026
La Biodola - Isola d'Elba

ITER NBI Systems



External Heating in ITER

• ITER will use several external heating methods at the same time:

- **Electron cyclotron resonance heating system**
- **Ion cyclotron resonance heating system**

Neutral beam injection system

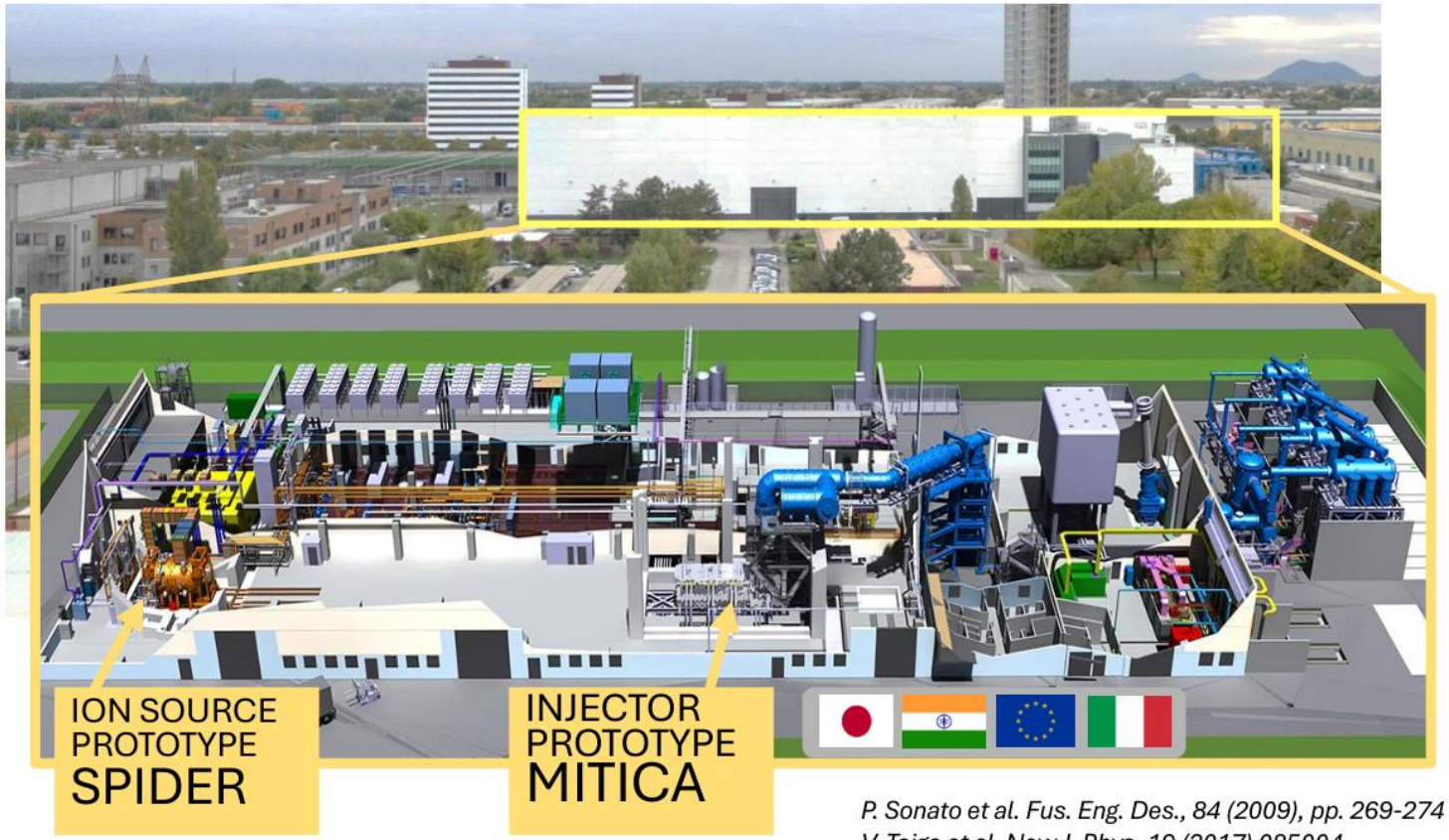
• ITER will use **two heating neutral beam injectors**:

- Each provides **16.5 MW** of heating power.
- Space is reserved for a possible third injector.

• A separate **Diagnostic Neutral Beam Injector**.

ITER Neutral Beam Test Facility

The ITER Neutral Beam Test Facility (NBTF) serves as a crucial testing ground for the development and validation of neutral beam injection systems essential for ITER's fusion power plant.



P. Sonato et al. Fus. Eng. Des., 84 (2009), pp. 269-274
V. Toigo et al. New J. Phys. 19 (2017) 085004

SPIDER (Source for Production of Ion of Deuterium Extracted from Rf Plasma)

- Focuses on the development and optimization of the ion source.
- Responsible for producing and accelerating the deuterium ions.
- Serves as a prototype for the ion source planned for use in ITER.

MITICA (Megavolt ITER Injector and Concept Advancement)

- Adds to the ion source technology by integrating high-energy beam acceleration.
- Aims to demonstrate the full-scale neutral beam injection system.
- Will be utilized in ITER for plasma heating, diagnostic, and control.

From a control-system perspective, these experiments are extremely demanding, involving several distributed servers, synchronized actions, real-time coordination, fault management, and continuous monitoring.

Importance of the Dispatcher System

A dispatcher system is critical for managing a complex, distributed environment of an experiment. Its primary roles include:

- **Centralized Supervision and Coordination:** It supervises the execution of actions across various actors, ensuring that the entire system follows the experimental sequence.
- **Synchronization of Distributed Servers:** It ensures that multiple action server instances coordinate their work, specifically making sure all actions with a lower sequence number are completed before moving to the next number.
- **Effective Load Sharing:** It allows multiple instances of the same server class to share the workload by picking actions from a shared list, ensuring tasks are distributed rather than concentrated on one server.
- **Dependency Management:** It handles the logic for "triggered" or conditional actions, ensuring they only execute once their specific prerequisites (dependencies) are met.
- **System Safety and Oversight:** It provides vital mechanisms for monitoring **action timeouts** and processing **abort requests** if a task becomes jammed or an experiment needs to stop.
- **Operation Monitoring:** Provides operators with **real-time visibility** into the status of every action across the distributed network

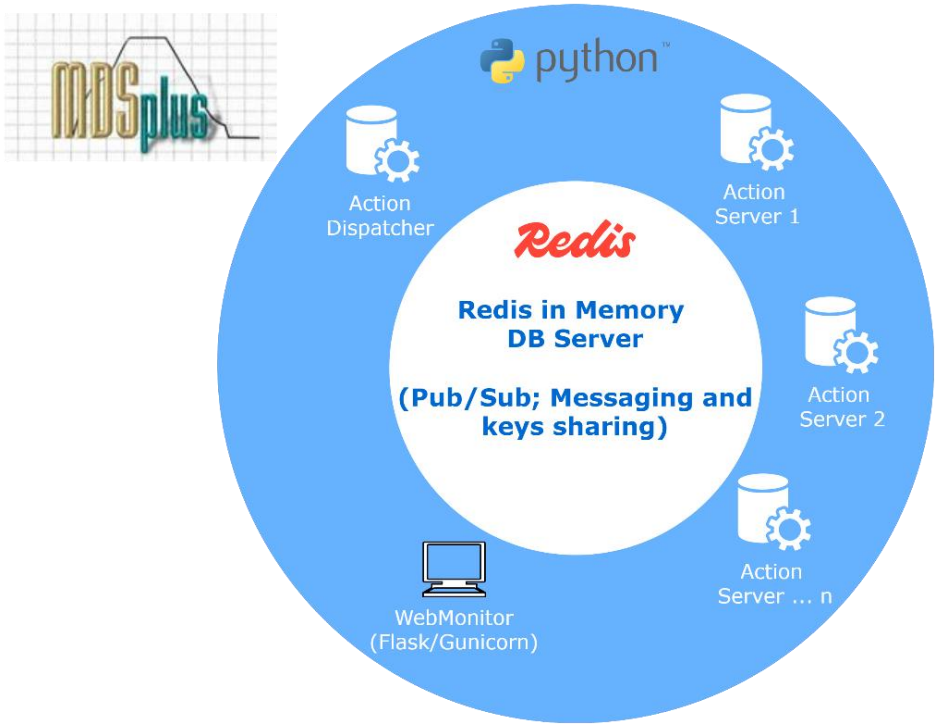
Limitations of the Previous Dispatcher System

The new MDSplus Redis Dispatcher was developed to address several bottlenecks and functional gaps in MDSplus Dispatcher:

- **Blocking on Asynchronous Actions:** In the previous system, the dispatcher would wait for the termination of actions tagged as asynchronous, which caused unnecessary delays in the experimental workflow.
- **Centralized Dispatcher Dependency:** The old system relied on a central dispatcher entity to manage all tasks. The new architecture is **"dispatchless“,** meaning action servers are now more autonomous.
- **Lack of Native Streaming Support:** The previous dispatcher supported **classic action execution**. Streamed actions—which allow for continuous method loops (init, step, finish)—were introduced with the new Redis-based version.
- **Synchronization and Race Conditions:** The new implementation specifically addresses potential **race conditions** using the REDIS mechanism to ensure that multiple servers do not attempt to execute the same action at the same time.
- **Improved Monitoring Technology:** Moved to a web-based Web Monitor improving real-time, browser-accessible, and highly reactive status updates.

New NBTF Dispatcher System

- **MDSplus acts as the data backbone**, storing experiment data, signals, and configuration information.
- **Action Dispatcher coordinates execution**, deciding which actions must run and in what order, **triggers tasks** to the available Action Servers and manages execution flow.



- **Redis is used as an in-memory coordination hub** for fast communication between components.
  - **Redis Pub/Sub handles event distribution**, so updates and triggers can be broadcast in real time.
  - **Redis shared keys provide lightweight state sharing**, allowing components to exchange status, flags, and control values.

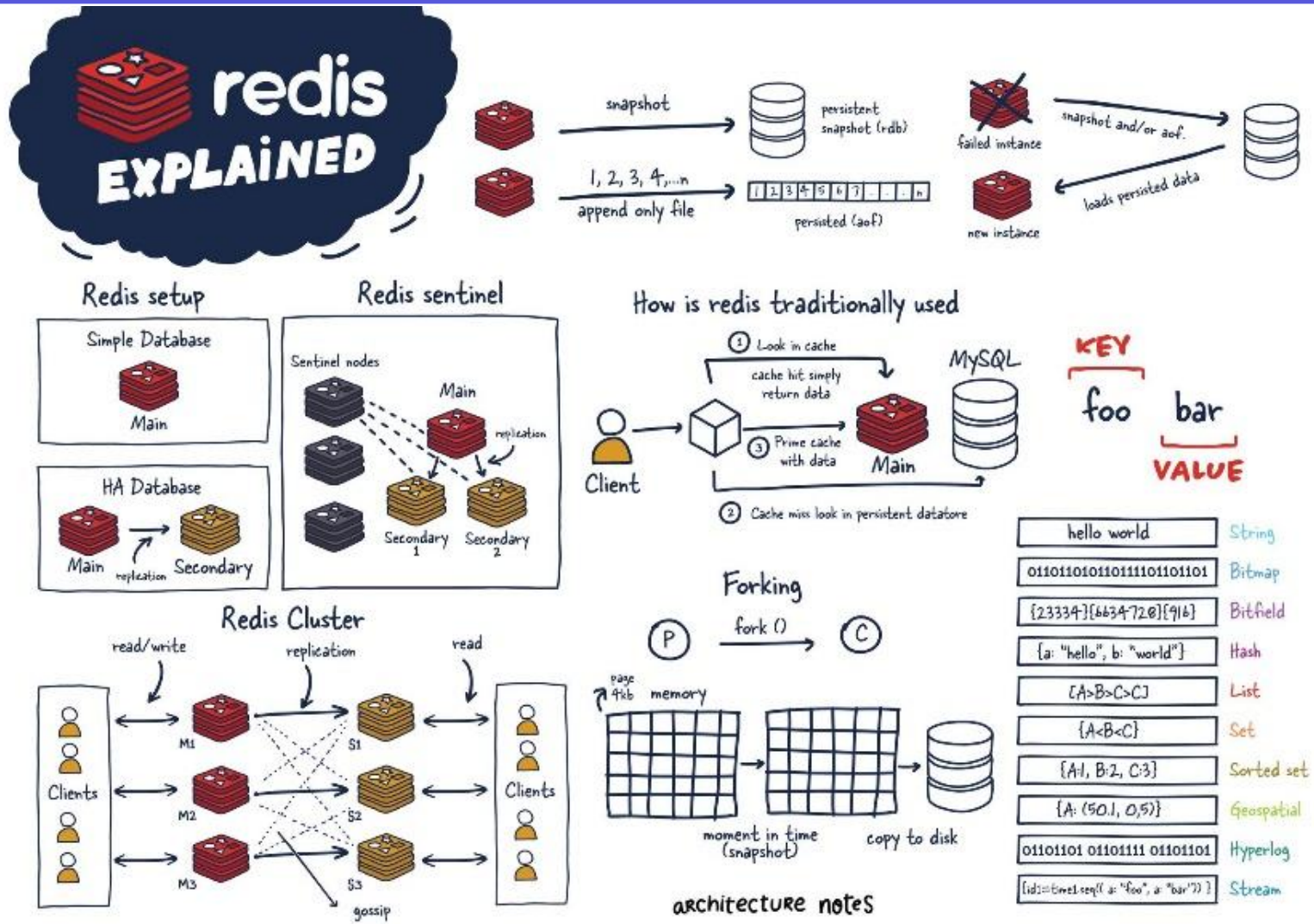
- **Action Servers execute the workload**, with multiple workers available for parallel or distributed processing.

- **WebMonitor provides a live operational view** through a Flask/Gunicorn web interface.

- **Scalable architecture**, with additional Action Servers added as the workload grows.

- **Improved responsiveness, flexibility, and maintainability:**
  - **Design separates control, messaging, and execution,**
  - **All environment developed in Python**

Redis in Memory Database



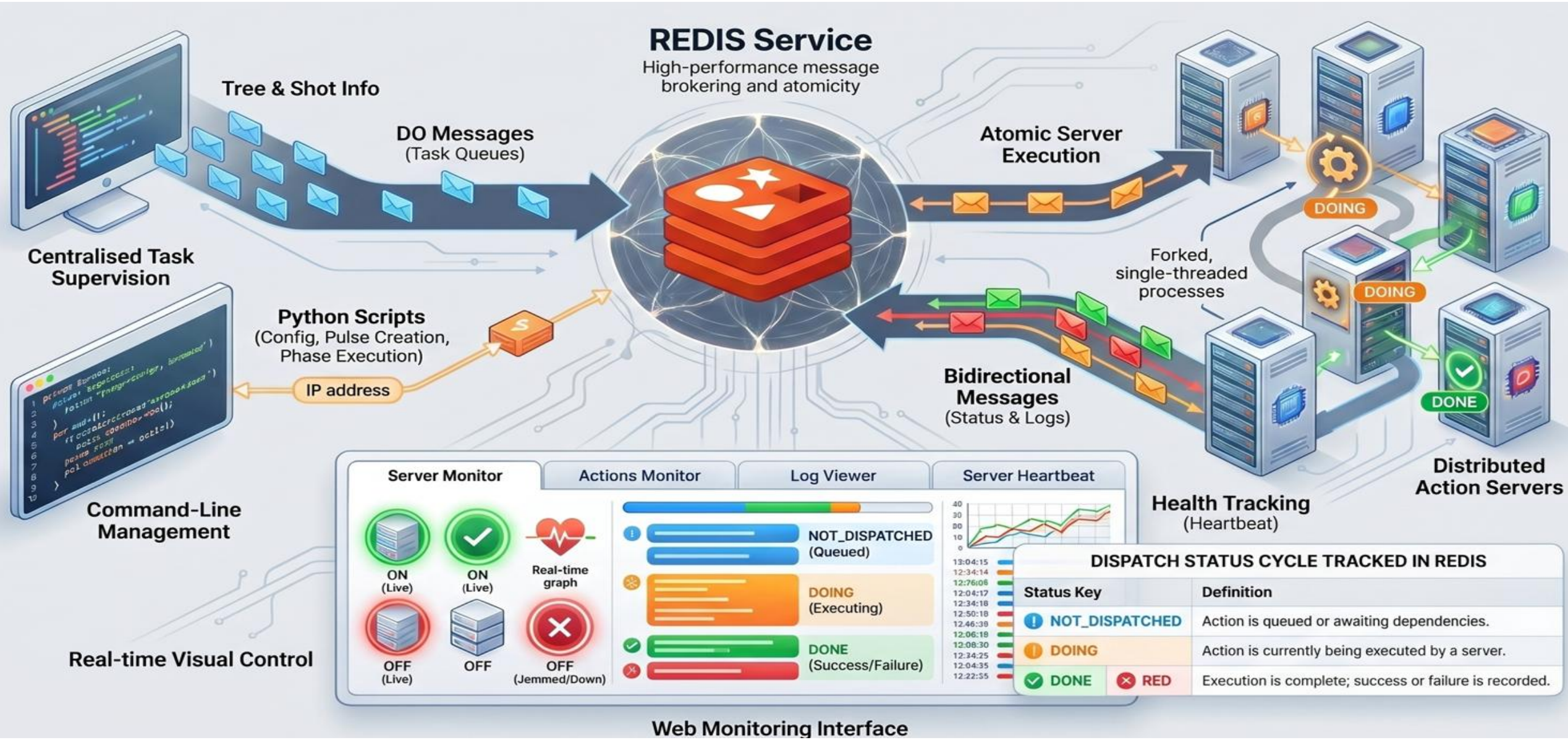
The Redis powerful capacity of handling data explained. Courtesy of Mahdi Yusuf (https://architecturenotes.co/redis/)

Redis architecture fits the needs of real-time distributed control systems:

- In-memory operation provides low latency and high throughput
- Supports data structures (hashes, lists, sets, streams) for:
  - Task queues
  - System states
  - Priorities
  - Data synchronization
- Redis Pub/Sub - efficient event-driven communication
- Persistence and recovery mechanisms - robustness and fast failure recovery
- Lightweight design and strong Python integration simplified development

Redis acts not only as a message broker, but as the synchronization and coordination backbone of the dispatcher framework

MDSplus Redis Dispatcher Architecture



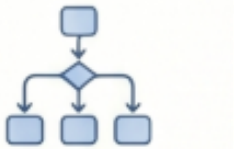
DISPATCH STATUS CYCLE TRACKED IN REDIS

Status Key		Definition
NOT_DISPATCHED		Action is queued or awaiting dependencies.
DOING		Action is currently being executed by a server.
DONE	RED	Execution is complete; success or failure is recorded.

MDSplus Redis Dispatcher Architecture

Action Dispatcher

Internal Table Building



Builds internal structures to identify all actions required for the specified tree, shot and phase.

Command Initiation

Processes DO_PHASE
Triggers the Action Servers execution using Redis PUB/SUB channel

Redis Server

PUB/SUB

ACTION_SERVER_PUBSUB

Signals servers to check task lists
Broadcasts DO message alert to all subscribed Action Servers

ACTION_DISPATCHER_PUBSUB

Signals serverscompletion feedback to the Dispatcher

HASHES

ACTION_INFO

Real-time updates to the current actions status

ACTION_LOG

Stores the latest screen output STDOUT of executed actions for log monitoring

LISTS

ACTION_SERVER_TODO

Queue of actions for a server class
Ensures tasks are popped atomically after PUB/SUB notification to servers.

Action Servers

Action Server Execution

Atomic Task Retrieval
After receiving DO signal repeteadly pops tasks from to do list until empty.

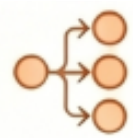
Forked Process Execution

Actions are executed by forked, sigle threaded processes with SIGALAR managing timeouts

Feedback and System Update

Update State Hashes and Completion Info
Publishes essage with tree shot, status and logs after child process completion.

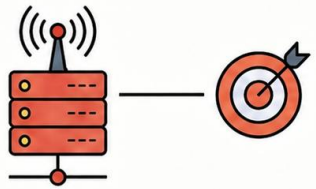
Triggers Dependent Actions



Automatically dispatches sequences for dipendent actions when the completed action was a prerequisite.

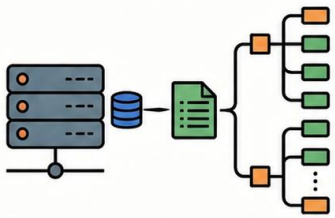
Building Action Dispatcher Tables

REDIS Sends BUILD_TABLES Command



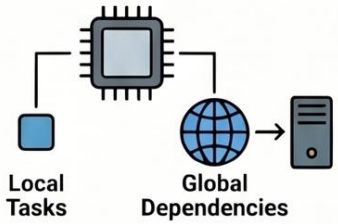
The command is broadcast via **REDIS** pubsub channels to the Action Dispatcher.

Server Reads MDSplus Tree



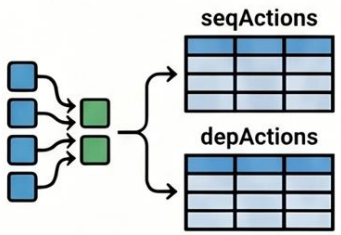
The **server accesses the MDSplus Tree** to collect all relevant action and timing information.

Map Local and Global Triggers



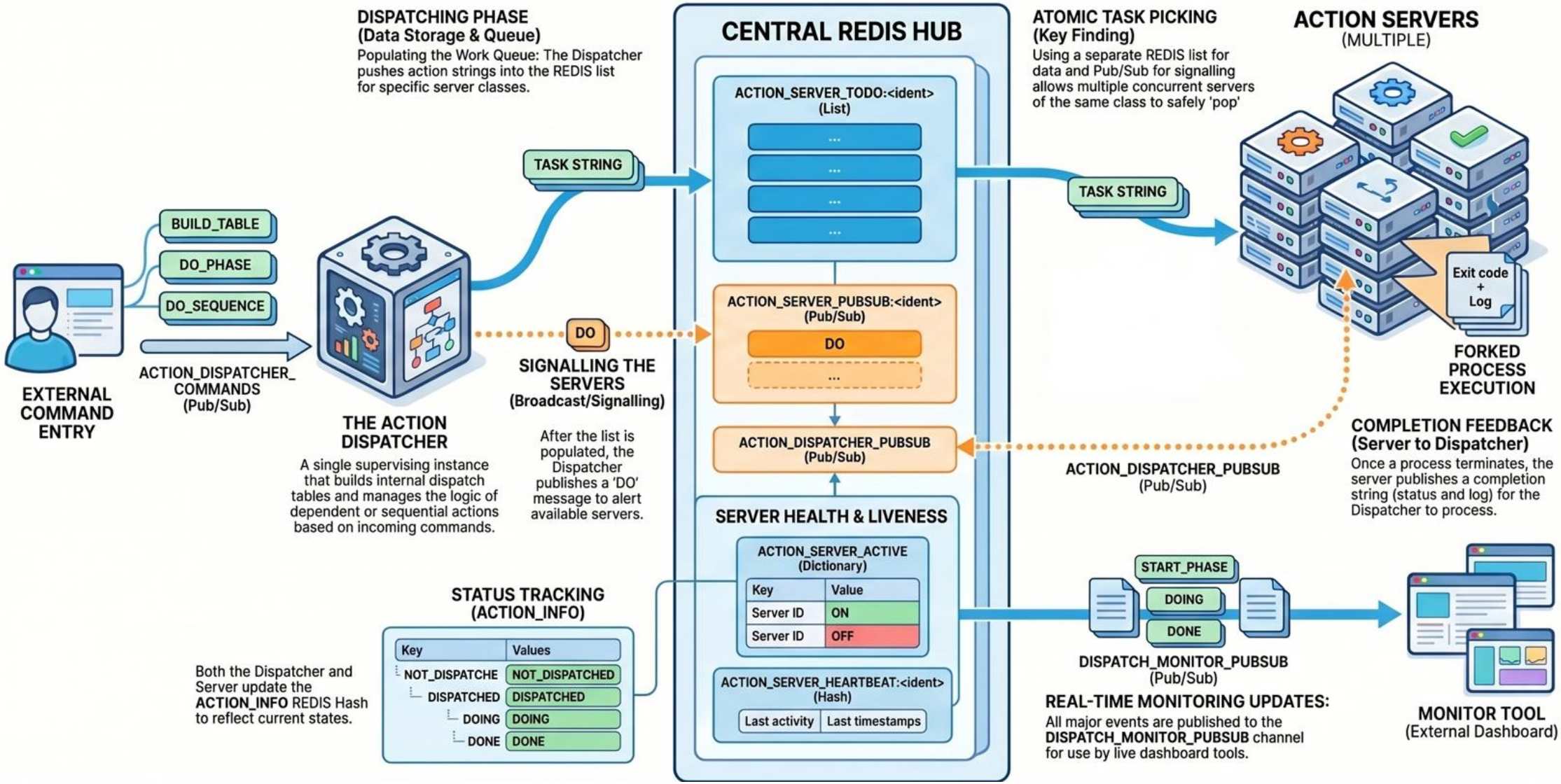
The system identifies **local tasks** and **dependencies** that will trigger other servers.

Populate Internal Python Tables

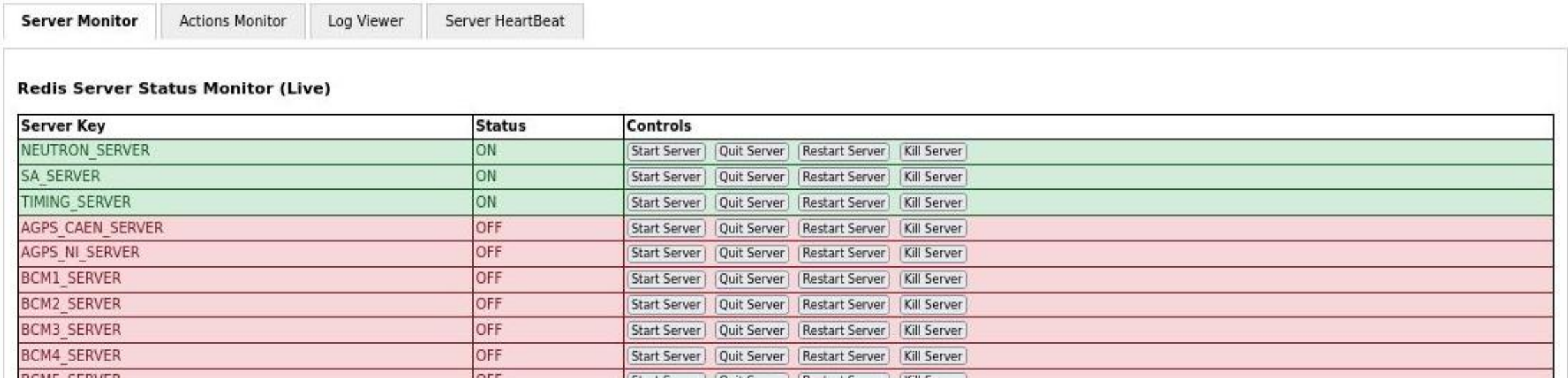


Data is stored in structures like **seqActions** and **depActions** to manage future execution steps.

MDSplus Redis Dispatcher Execution Flow



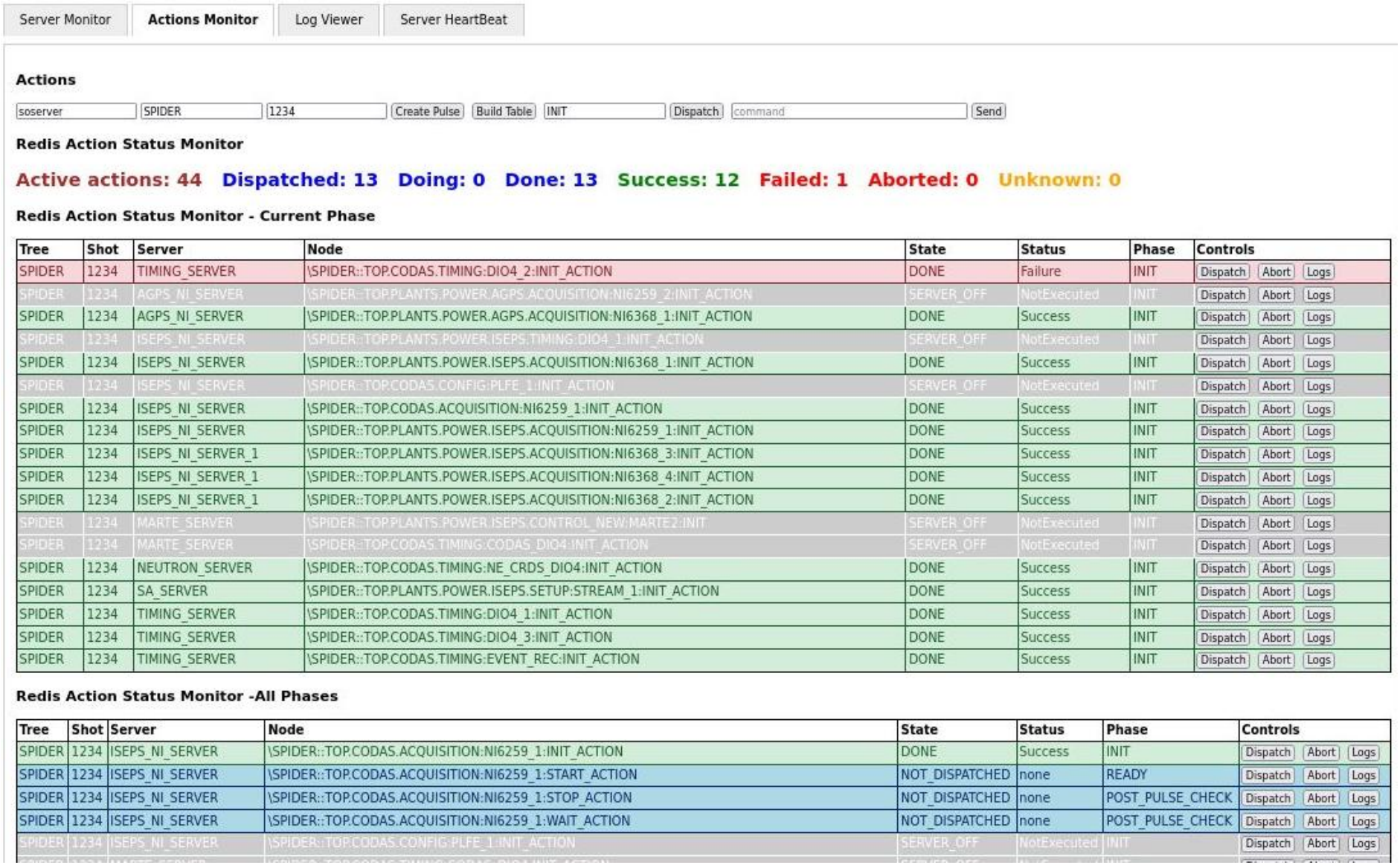
MDSplus Redis Dispatcher WebMonitor



Server Monitor

- Real-time view of registered action servers
- Displays server ON/OFF state
- Automatic live status refresh
- Active servers prioritized at the top
- Start, restart, quit and kill controls from the interface

MDSplus Redis Dispatcher WebMonitor



Tree	Shot	Server	Node	State	Status	Phase	Controls
SPIDER	1234	TIMING_SERVER	\SPIDER::TOP.CODAS.TIMING:DIO4_2:INIT_ACTION	DONE	Failure	INIT	Dispatch Abort Logs
SPIDER	1234	AGPS_NI_SERVER	\SPIDER::TOP.PLANTS.POWER.AGPS.ACQUISITION:NI6259_2:INIT_ACTION	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs
SPIDER	1234	AGPS_NI_SERVER	\SPIDER::TOP.PLANTS.POWER.AGPS.ACQUISITION:NI6368_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.PLANTS.POWER.ISEPS.TIMING:DIO4_1:INIT_ACTION	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.PLANTS.POWER.ISEPS.ACQUISITION:NI6368_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.CONFIG:PLFE_1:INIT_ACTION	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.ACQUISITION:NI6259_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.PLANTS.POWER.ISEPS.ACQUISITION:NI6259_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER_1	\SPIDER::TOP.PLANTS.POWER.ISEPS.ACQUISITION:NI6368_3:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER_1	\SPIDER::TOP.PLANTS.POWER.ISEPS.ACQUISITION:NI6368_4:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER_1	\SPIDER::TOP.PLANTS.POWER.ISEPS.ACQUISITION:NI6368_2:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	MARTE_SERVER	\SPIDER::TOP.PLANTS.POWER.ISEPS.CONTROL_NEW:MARTE2:INIT	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs
SPIDER	1234	MARTE_SERVER	\SPIDER::TOP.CODAS.TIMING:CODAS_DIO4:INIT_ACTION	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs
SPIDER	1234	NEUTRON_SERVER	\SPIDER::TOP.CODAS.TIMING:NE_CRDS_DIO4:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	SA_SERVER	\SPIDER::TOP.PLANTS.POWER.ISEPS.SETUP:STREAM_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	TIMING_SERVER	\SPIDER::TOP.CODAS.TIMING:DIO4_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	TIMING_SERVER	\SPIDER::TOP.CODAS.TIMING:DIO4_3:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	TIMING_SERVER	\SPIDER::TOP.CODAS.TIMING:EVENT_REC:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs

Tree	Shot	Server	Node	State	Status	Phase	Controls
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.ACQUISITION:NI6259_1:INIT_ACTION	DONE	Success	INIT	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.ACQUISITION:NI6259_1:START_ACTION	NOT_DISPATCHED	none	READY	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.ACQUISITION:NI6259_1:STOP_ACTION	NOT_DISPATCHED	none	POST_PULSE_CHECK	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.ACQUISITION:NI6259_1:WAIT_ACTION	NOT_DISPATCHED	none	POST_PULSE_CHECK	Dispatch Abort Logs
SPIDER	1234	ISEPS_NI_SERVER	\SPIDER::TOP.CODAS.CONFIG:PLFE_1:INIT_ACTION	SERVER_OFF	NotExecuted	INIT	Dispatch Abort Logs

Actions Monitor

- Monitors action execution in real time
- Groups actions by tree and shot

Action Controls

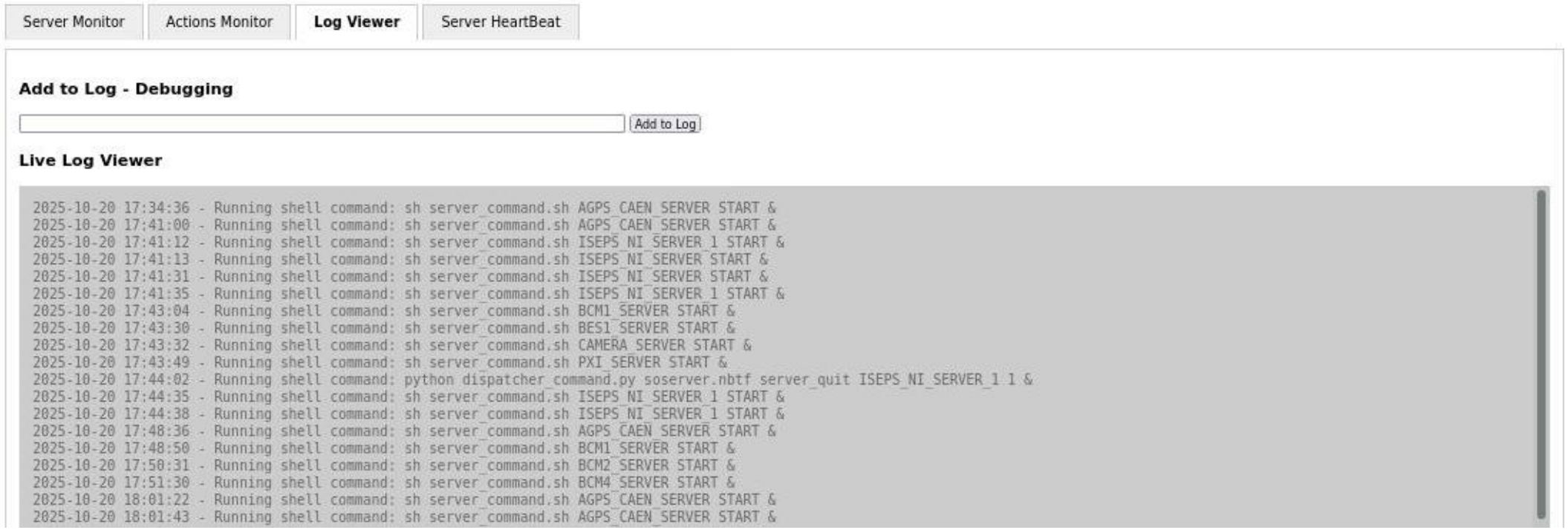
- Dispatch and Abort individual actions
- Inspect execution logs stored in Redis
- Send global Commands, Dispatch and Build Tables

Colour-Coded Status Indicators

Live Summary Panel

- Failed actions
- Aborted actions
- Success
- Overall execution progress

MDSplus Redis Dispatcher WebMonitor



Log Viewer

- Real-time application log streaming
- Centralized view of runtime logs
- Submit test log messages for debugging
- Supports additional server-side log integration
- Useful for debugging and system verification

Technical Stack of MDSplus Redis Dispatcher

Node.js and React were replaced by Flask for a unified Python technology stack that helps improving long-term maintainability and deployment simplicity.

Summary & Conclusions – Dispatcher Implementation

- **Shift to a “dispatchless" Architecture:** The system has transitioned from a fully centralized dispatcher to a more distributed model where autonomous **Action Servers** derive their own execution logic.
- **REDIS-Powered Synchronization:** The architecture leverages **REDIS in-memory hashes** for real-time status tracking and **Pub/Sub channels** for efficient command broadcasting across distributed actors.
- **Atomic Concurrency Control:** The implementation of the **REDIS "watch" mechanism** ensures thread safety and prevents race conditions, allowing multiple server instances to share the workload without duplicating tasks.
- **Support for Streamed Actions:** The new system introduces native support for **streamed actions** (utilizing init, step, and finish method loops), expanding capabilities beyond the classic execution modes.
- **Improved Experimental Efficiency:** By removing the requirement to wait for the termination of **asynchronous actions** before ending a phase, the new dispatcher reduces idle time during experiments.
- **Enhanced Real-Time Visibility:** The **WebMonitor** provides operators with a comprehensive, browser-based interface for monitoring action statuses, server heartbeats, and live execution logs.
- **Scalability and Robustness:** The decentralized design and load-sharing capabilities ensure the system is highly scalable and capable of handling the complex requirements of large-scale facilities like NBTF

Summary & Conclusions – System Integration

- **Replacement of the legacy centralized dispatcher with a scalable Redis-based distributed architecture**
  The work successfully introduced a modular and fault-tolerant MDSplus Action Dispatcher using Redis as the coordination and messaging backbone, improving reliability, scalability, and recovery capabilities for complex fusion experiments such as SPIDER and MITICA.
- **Commissioning of the new fully integrated real-time monitoring and control environment**
  A Python-native WebMonitor was created to provide live supervision of servers, actions, logs, and heartbeat status, enabling operators to monitor and control distributed execution in real time through a unified interface.
- **Successful validation in operational fusion environments with broader adoption potential**
  The new dispatcher framework was tested during **full discharge cycles in SPIDER and MITICA** and has already attracted adoption interest from **external laboratories such as MIT Plasma Science and Fusion Center**, demonstrating its applicability beyond ITER NBTF.

Acknowledgements

This work has been carried out within the framework of the **EUROfusion Consortium**, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

The work leading to this presentation has been funded partially by **ITER** under activity work programs AWP2020-2021. This publication reflects the views only of the authors and ITER Organization cannot be held responsible for any use, which may be made of the information contained therein. The views and opinions expressed herein do not necessarily reflect those of the ITER Organization.

IPFN activities were supported by **FCT - Fundação para a Ciência e Tecnologia**, I.P. by project reference UIDB/50010/2020 and DOI identifier 10.54499/UIDB/50010/2020 (https://doi.org/10.54499/UIDB/50010/2020), by project reference UIDP/50010/2020 and DOI identifier DOI 10.54499/UIDP/50010/2020 (https://doi.org/10.54499/UIDP/50010/2020) and by project reference LA/P/0061/202 and DOI 10.54499/LA/P/0061/2020 (https://doi.org/10.54499/LA/P/0061/2020).

Thank you for your attention