GINKGO
BIOWORKS

Control and Data Plane Integration for High-throughput Modular Automation

(3 Case Studies)

WWW.GINKGO.BIO

Control and Data Plane Integration for High-throughput Modular Automation (3 Case Studies)

Greg Lammers, Jose Cortez, Chris Bremner

INTRODUCTION

Ginkgo's Automation Control Software (ACS) provides a flexible interface for the integration of our RAC systems with a wide range of customer software integration needs. Our platform uses open-source messaging standards and schema definition tools to ensure that new integrations can be developed rapidly. This paper details three case studies where ACS was integrated into various high-throughput environments. Each case study describes a scenario presented and summarizes the solution that was developed both for launching protocol runs and integrating the data generated by them into automated LIMS data pipelines.

The three case studies are presented here in order of increasing complexity. We see the simpler integration presented in Case Study A as the integration mode that covers most high-throughput customers. Case Study A should be thought of as the default mode. Case Studies B and C are heavy-weight modes of integration that illustrate what software integrations can look like when sizable software and hardware ecosystems are already in place, such as a Manufacturing Execution System (MES) and data flow management from multiple workcell platforms.

We do not cover the low-throughput scenario (i.e. no automated data pipelines) in this document, which can easily be accomplished without the need for any customized software. This scenario deserves mention here since it will suit the needs of many customers, but it is too simple to merit its own integration case study. Files generated by devices are made available in ACS applications, and they can be downloaded manually as needed. In Case Study A, we will point out the two relevant steps for this integration mode.

(Note: In this document, we'll use the general term MES instead of the more specialized term "Laboratory Execution System (LES)" that typically applies in this domain since MES is more widely known.)

CASE STUDY A — DIRECT PROTOCOL EXECUTION WITH FILE-BASED OUTPUT

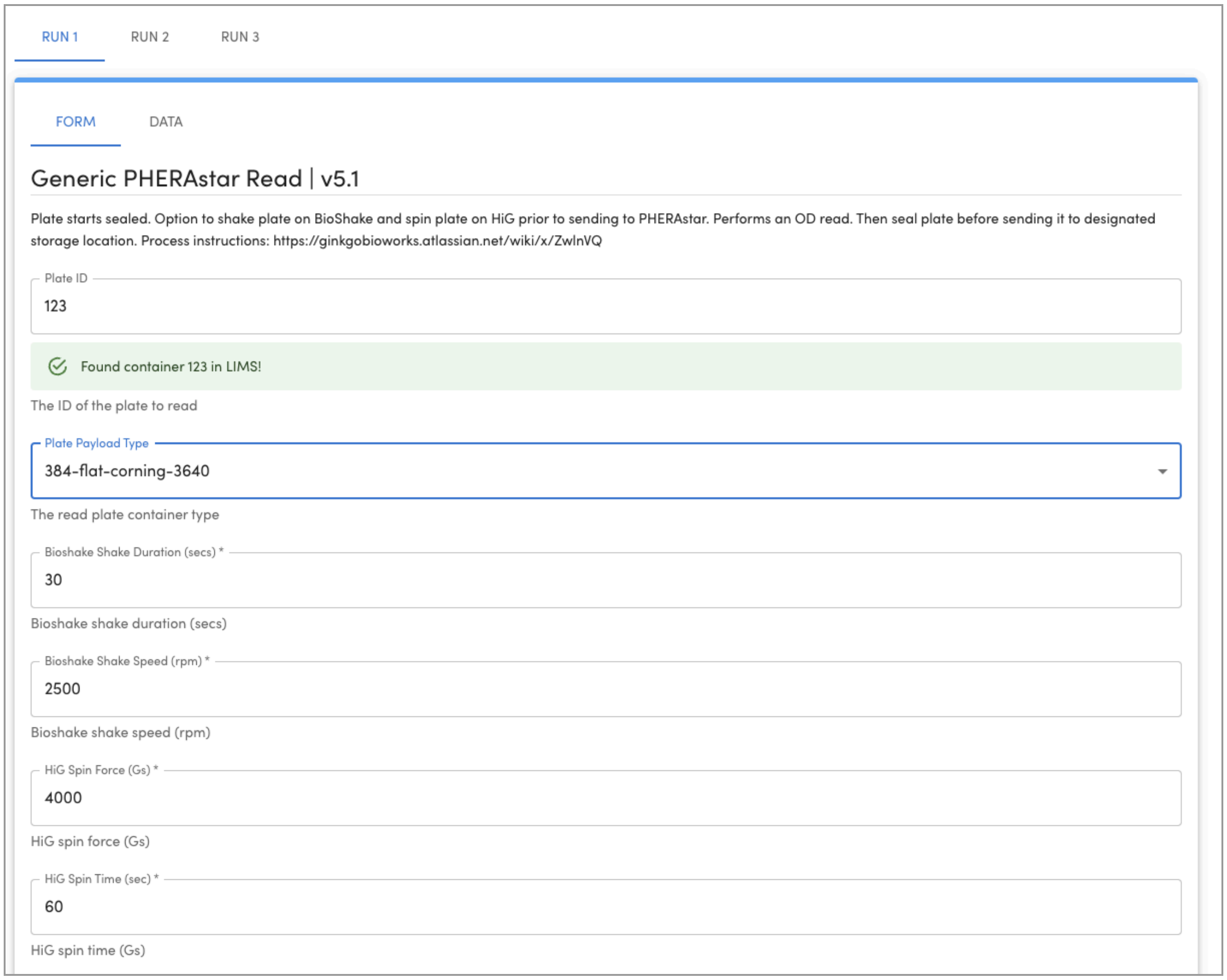
Scenario

In this customer’s scenario, laboratory scientists directly execute protocols on RACs without an orchestration layer, like a Manufacturing Execution System (MES). They expect to receive raw data from the instruments on the RACs, and they also expect this data to be available in a data warehouse.

Launching Protocol Runs

The RAC platform provides a tool that allows users to launch protocol runs directly on RACs without the need for external software. This tool provides a schema-driven, user-friendly interface for launching protocol runs with their required parameters. A screenshot of this tool is provided in **FIG. 1** for a plate read protocol on a BMG Labtech PHERAstar®.

For this case, the customer needed additional validation and transformation logic added to the protocol launching tool.



Generic PHERAstar Read | v5.1

Plate starts sealed. Option to shake plate on BioShake and spin plate on HiG prior to sending to PHERAstar. Performs an OD read. Then seal plate before sending it to designated storage location. Process instructions: https://ginkgobioworks.atlassian.net/wiki/x/ZwlnVQ

Plate ID: 123

Found container 123 in LIMS!

The ID of the plate to read

Plate Payload Type: 384-flat-corning-3640

The read plate container type

Bioshake Shake Duration (secs) *: 30

Bioshake shake duration (secs)

Bioshake Shake Speed (rpm) *: 2500

Bioshake shake speed (rpm)

HiG Spin Force (Gs) *: 4000

HiG spin force (Gs)

HiG Spin Time (sec) *: 60

HiG spin time (Gs)

FIGURE 1. **Protocol Launcher -** Generic PHERAstar Read form

This was provided as part of the agreement between the customer and Ginkgo Automation. The custom version of the protocol launcher is maintained by Ginkgo Automation separate from other customer code and is updated as the customer designs new protocols.

Data Return

This version of the RAC-based workflow requires scientists and data scientists to understand the output coming directly from devices installed on the RAC system. These devices typically generate files that are stored local to the device and are exposed through applications provided by ACS.

The primary data-producing instrument in this example scenario is a Pherastar plate reader. When the ACS application for the device completes an operation, it produces a message to its message bus with the following contents:

JavaScript
```javascript
{
  "data": {
    "file_url": "<ACS_APPLICATION_URL>/data/<UUID>.1569902.pherastar_results.csv",
    "event_type": "plate_read",
    "line_number": 3,
    "container_id": "id123",
    "protocol_name": "OD600"
  },
  "baggage": {
    "acs_app_version": "5.3.0",
    "protocol_run_id": "<UUID>",
    "protocol_run_step_id": "<UUID>"
  },
  "payloads": {
    "payload": {
      "id": "<UUID>",
      "type": "96-flat-corning-3370",
      "barcode": "1111111"
    }
  },
  "timestamp": "2024-08-26T18:52:01.199233+00:00",
  "acs_data_type": "biological_event",
  "recipe_run_id": "<UUID>",
  "submodule_name": "pherastar-sm-1",
  "protocol_run_id": "<UUID>"
}
```

The customer in this scenario used custom software to listen to the message bus and wait for events that matched this `plate_read` event type. This custom software initiated an in-house data pipeline by retrieving the data file from the plate reader through the ACS Application API and placing it in an AWS S3 bucket using a naming convention to record metadata about the operation. The components involved in this flow are depicted in **FIG. 2**.

The ACS message bus is built on top of the open-source Apache Kafka ® project, a widely used messaging platform. This technology has client libraries available for almost all common programming languages. In this scenario, the customer provided requirements so that Ginkgo Automation could develop the custom component using the confluent-kafka library for Python to write a simple Kafka consumer. This consumer processes messages by deserializing them according to the JSON format defined in the Confluent Schema Registry that runs as part of ACS. Once deserialized, messages that match the format of events related to the Pherastar plate reader are passed to a message handler that retrieves relevant data files from the ACS Pherastar application using a RESTful API. In this example, the API endpoint for the file contents is provided in the following message where `ACS_APPLICATION_URL` is a base URL for the application.

```None
{ACS_APPLICATION_URL}/data/<UUID>.1569902.pherastar_results.csv
```

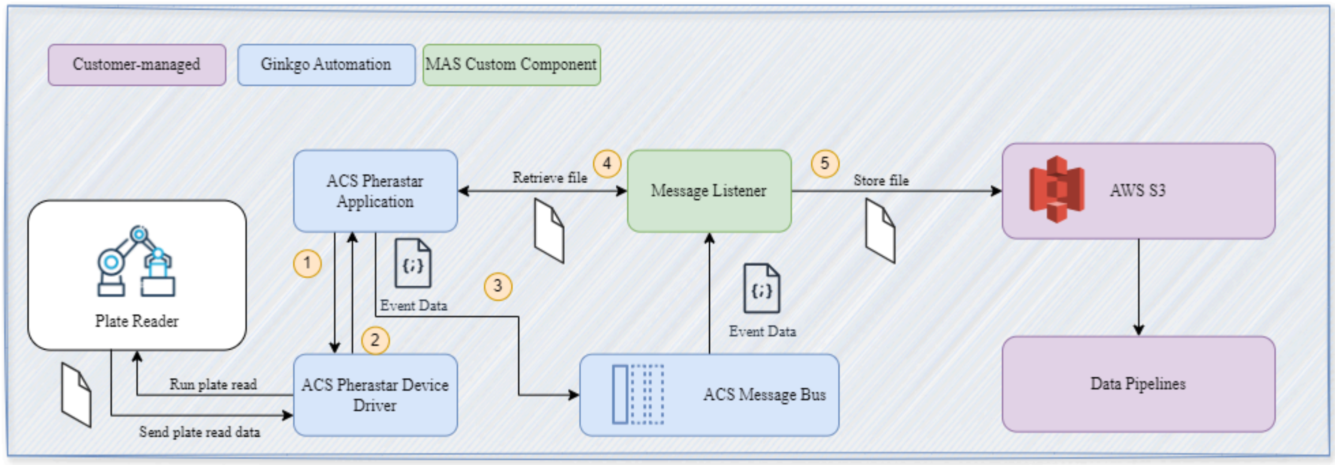


FIGURE 2. Case Study A — High Level Architecture 1) Plate read is initiated on the Pherastar device. 2) Once the plate read is completed, the data is returned to the ACS Pherastar Application. 3) ACS Pherastar Application publishes a Kafka message indicating a new data file is available. 4) Custom listener that is subscribed to the Kafka topic receives notification that the new data file is available and retrieves the file from the ACS Pherastar application. 5) Custom listener forwards the file it retrieved to AWS S3 for further downstream processing.

The contents of the result file are then uploaded to AWS S3 using the boto3 library for Python. The customer then manages the data analysis pipeline using their own in-house tools and transformation.

(Note: For customers with lower throughput needs (i.e. without automated data pipelines), only steps 1 and 2 would be relevant for their scenario. They could directly download the data from the device via the ACS Application.)

Summary

For customers with an interest focused on the raw results from instruments on RACs managed by ACS, Ginkgo Automation provides out-of-the-box tools to create useful data pipelines quickly. Additionally, simple custom code components enable customized protocol launching or data return via our APIs.

CASE STUDY B — MANUFACTURING EXECUTION SYSTEM (MES) INTEGRATION WITH AUTOMATION CONTROL SOFTWARE (ACS)

Scenario

In this scenario, laboratory scientists execute protocols on RACs following the guidance of a Manufacturing Execution System (MES). These operators are responsible for preparing the protocol runs, but they are not the ones doing the data analysis. Instead, a separate team of scientists must review the data from these experiments in the company's custom LIMS software. Additionally, the company's LIMS performs extensive sample lineage tracking.

Launching Protocol Runs

Because this customer extensively used MES software to orchestrate its lab workflows, they needed to integrate RACs workloads with the MES via the ACS API. The customer elected to design its own protocols and forms for parameterizing them, so they simply needed a way to submit a request and monitor the progress of that request.

The customer chose to quickly develop a message producer that could submit protocol run requests via the ACS's messaging API using a published schema. There's also the option of launching protocols via ACS's RESTful API, as we'll see with Case Study C.

An example of a message adhering to this schema is shown here:

```javascript
{
    "baggage": {
        "acs_protolaunch_env": "virtual",
        "acs_protolaunch_version": "2.10.4"
    },
    "priority": null,
    "protocol_name": "generic_send_to_storage",
    "protocol_parameters": "{\"storage_submodule\": \"steristore-vsm-3\",
\"storage_location\": \"steristore-10-position\", \"plate_id\": \"123\",
\"additional_step\": \"\", \"plate_payload_type\": \"\", \"lid_submodule_type_name\":
\"\", \"relid_source\": true, \"pl_seal_type\": \"alu\"}",
    "protocol_run_id": "<UUID>",
    "requesting_user": {
        "string": "ginkgo_user"
    },
    "timestamp": "2024-01-01T16:59:41.102967+00:00",
    "version_specifier": null
}
```

The baggage field here deserves special mention. Any contents passed into this field will be passed along throughout the protocol run, allowing application integrators to develop their own additional semantic layer, such as sample context, on top of the protocol run itself.

The protocol_name field is used to specify the operation that the automation platform should perform. Each defined protocol specifies a schema for parameters that can be provided at runtime, and these are stringified as JSON and passed in the protocol_parameters field. The protocol_run_id is provided by the caller and uses the UUID4 format to ensure that all protocol runs receive a unique identifier.

The customer also developed a Kafka consumer, integrated into their MES, to monitor the status of runs. The ACS software that manages RACs emits events on a Kafka topic whenever an operation starts or completes.

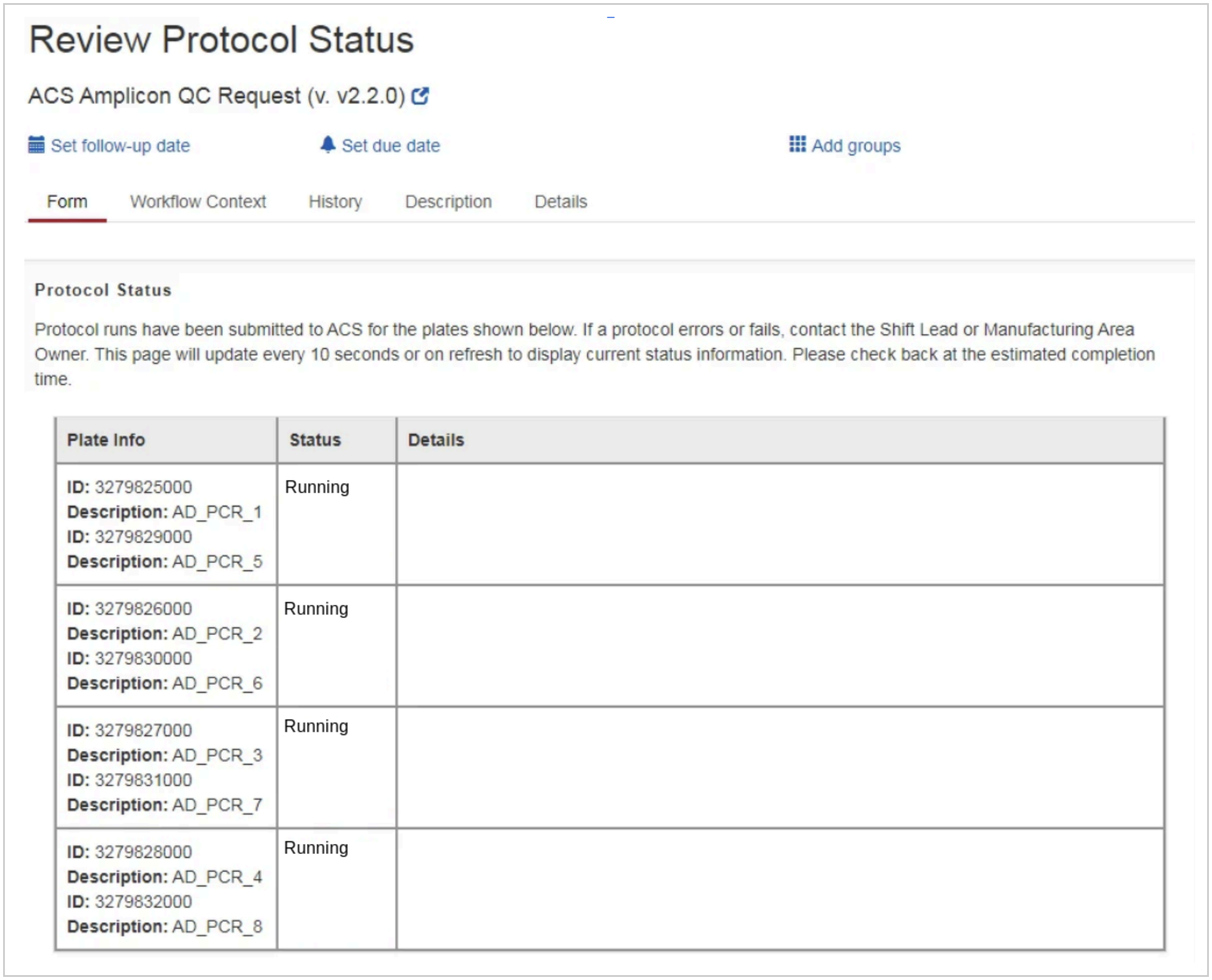
An example of this type of message for the completion of a protocol run is shown below:

```
JavaScript
{
    "baggage": {
        "acs_protolaunch_env": "virtual",
        "acs_protolaunch_version": "2.10.4"
    },
    "error_message": null,
    "in_error": false,
    "module": "acs-dev-system",
    "payloads": [
        "123"
    ],
    "protocol_name": "generic_send_to_storage",
    "protocol_parameters":
"{\"plate_id\":\"123\",\"pl_seal_type\":\"alu\",\"relid_source\":true,\"additional_ste
p\":\"\",\"storage_location\":\"steristore-10-position\",\"storage_submodule\":\"steri
store-vsm-3\",\"plate_payload_type\":\"\",\"lid_submodule_type_name\":\"\"}",
    "protocol_run_id": "<UUID>",
    "requested_timestamp": "2024-01-01T16:59:41.102967+00:00",
    "requesting_user": {
        "string": "ginkgo_user"
    },
    "status": "Finished",
    "timestamp": "2024-01-02T17:00:49.883807+00:00"
}
```

The customer designed custom web forms in their MES for each protocol they designed to run on their RAC system. They created handlers to parse this form data, translate it into the format specified by ACS, and submit a protocol run request using the required Kafka topic. They also created a form to monitor the progress of protocol runs and maintain the state of the workflow orchestration, enabling transparent monitoring of the automation platform. This form is illustrated in **FIG. 3**.

Data Return

The customer in this scenario leveraged their MES in order to facilitate data gathering, sample tracking, and downstream analysis. The MES software was programmed to wait for messages from ACS signifying that a protocol run was complete. Upon receipt of this message, the MES software triggered the next steps in its defined workflow, which often included retrieving and copying data from ACS Applications.



Review Protocol Status

ACS Amplicon QC Request (v. v2.2.0)

Protocol Status

Protocol runs have been submitted to ACS for the plates shown below. If a protocol errors or fails, contact the Shift Lead or Manufacturing Area Owner. This page will update every 10 seconds or on refresh to display current status information. Please check back at the estimated completion time.

Plate Info	Status	Details
ID: 3279825000 **Description:** AD_PCR_1 **ID:** 3279829000 **Description:** AD_PCR_5	Running	
ID: 3279826000 **Description:** AD_PCR_2 **ID:** 3279830000 **Description:** AD_PCR_6	Running	
ID: 3279827000 **Description:** AD_PCR_3 **ID:** 3279831000 **Description:** AD_PCR_7	Running	
ID: 3279828000 **Description:** AD_PCR_4 **ID:** 3279832000 **Description:** AD_PCR_8	Running	

FIGURE 3. Customized Protocol Status Monitoring in the MES

As an example, upon completion of a liquid transfer protocol run on an Echo Liquid Handler, the MES software triggered custom code to perform the intended liquid transfers in their LIMS. Notably, this customer relied on the intended liquid handling operation to determine which transfers to model in software; they did not need to parse logs from instruments to verify against execution records.

Summary

Customers who already use a workflow orchestration system like an MES can integrate with ACS and RACs by building around the ACS message bus to submit and monitor protocol run requests. The data return pipeline can be customized as needed, as all data produced by instruments managed by ACS is available through ACS application APIs.

CASE STUDY C — DEEP INTERNAL PLATFORM INTEGRATION

Scenario

In the third scenario, we present the fully integrated software solution used by Ginkgo Bioworks for internal automation projects. Ginkgo leverages a portfolio of its in-house automation platform combined with offerings from third-party automation system vendors in order to complete its own automation workflows. These disparate automation platforms are united through a distributed system that manages the creation of automation runs as well as the processing of data produced by the various platforms.

Launching Protocol Runs

There are two user personas who launch protocol runs at Ginkgo: process development engineers who design protocols and automation operators who execute experiments defined in an in-house workflow orchestration system.

Process development engineers who must rapidly iterate on protocols take advantage of the included ACS Protocol Launcher tool shown in **FIG. 4**. This tool allows them to define the allowed inputs for protocols via a no-code UI or programmatically via JSON schema. In addition to defining inputs, this tool also allows for launching batches of runs with different parameter combinations, speeding up the process of protocol development.

As a process becomes more rigidly defined, Ginkgo process engineers may elect to define it using an in-house workflow engine. They can use ACS's RESTful API to enable custom forms in their workflow engine that operators can use to launch protocols on RAC systems.

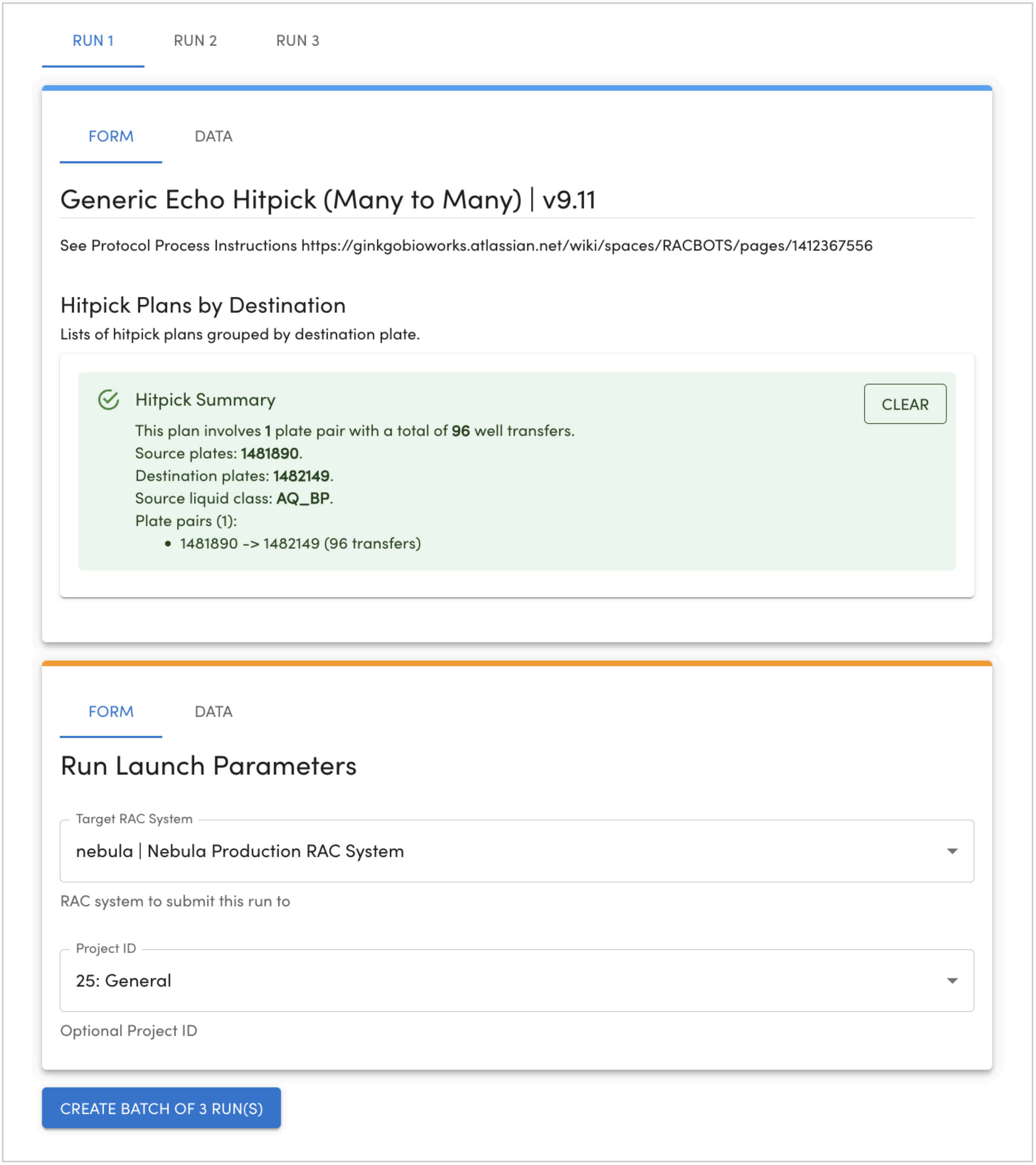
Data Return

Ginkgo Automation has developed an automation event data integration system, named the Event Processing Pipeline (EPP), for connecting in-house and any third party automation system. To connect ACS to Ginkgo's software ecosystem via EPP, new adapter components, called Event Extractors (shown below) were added. A high-level view of the integration is shown in **FIG. 5**.

Briefly, EPP consists of three major subsystems:

- An Event Store that records all events that occur on an automation platform and offers a way to query for those events in a standardized format, as well as a way to record the processing status of those events
- Extractors that read events from an automation platform and transform them into a standard representation that can be passed to the Event Store

- An event processor that retrieves the standardized events, processes them using custom code, and then updates the processing status of those events in the Event Store.



Generic Echo Hitpick (Many to Many) | v9.11

See Protocol Process Instructions https://ginkgobioworks.atlassian.net/wiki/spaces/RACBOTS/pages/1412367556

Hitpick Plans by Destination

Lists of hitpick plans grouped by destination plate.

Hitpick Summary

This plan involves **1** plate pair with a total of **96** well transfers.
Source plates: **1481890**.
Destination plates: **1482149**.
Source liquid class: **AQ_BP**.
Plate pairs (1):

- 1481890 -> 1482149 (96 transfers)

Run Launch Parameters

Target RAC System: nebula | Nebula Production RAC System

RAC system to submit this run to

Project ID: 25: General

Optional Project ID

FIGURE 4. Protocol Launcher - Generic Echo Hitpick form

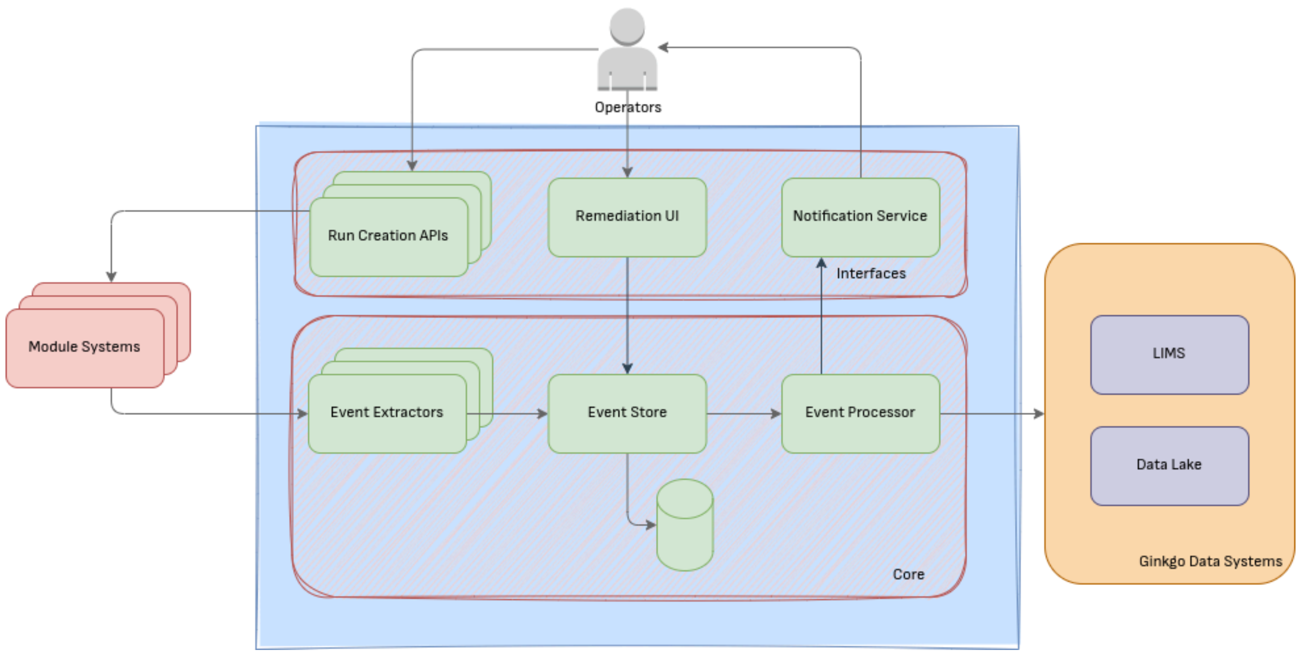


FIGURE 5. **EPP High Level Architecture**

Due to the standardization of event representations, onboarding of RAC onto this system only required developing new extractors to read the events emitted by ACS. These extractors were implemented as Kafka consumers that read from an ACS 'omnibus' Kafka topic, which records events in the strict order that they occur on instruments controlled by the system. As an example, consider the case of a liquid transfer performed by an Agilent Bravo® liquid handler. When this device completes an operation, ACS publishes a message to the Kafka topic with the following contents:

```
JavaScript
{
      "data": {
        "volume": "5",
        "event_type": "stamp",
        "tool_shape": "SBS96",
        "line_number": "6",
        "source_well": "B2",
        "source_plate": "123456",
        "destination_well": "A1",
        "destination_plate": "234567"
      },
      "baggage": {
        "acs_app_version": "5.3.0",
        "protocol_run_id": "<UUID>",
        "acs_broker_version": "4.11.0",
        "protocol_run_step_id": "<UUID>",
      },
      "payloads": {
        "tips": {
          "id": "<UUID>",
          "type": "axygen-20-bravo-96",
          "barcode": ""
        },
        "source": {
          "id": "<UUID>",
          "type": "384-well Plate Echo PP",
          "barcode": "1111111"
        },
        "destination": {
          "id": "<UUID>",
          "type": "96-round-axygen-pdw11cs-halfdeep",
          "barcode": "222222"
        }
      },
      "timestamp": "2024-09-09T22:42:03.257853+00:00",
      "module_name": "nebula",
      "acs_data_type": "biological_event",
      "recipe_run_id": "<UUID>",
      "submodule_name": "bravo-96-sm-1",
      "protocol_run_id": "<UUID>"
    },
    "automation_iso_timestamp": "2024-09-09T22:42:03.257853+00:00",
    "automation_system_details": {
      "system_type": "acs",
      "service_versions": [
        {
          "name": "acs_app_version",
          "version": "5.3.0"
        },
        {
          "name": "acs_broker_version",
          "version": "4.11.0"
          }
        ]
      }
    }
```

This is then consumed by an extractor reading from the topic and transformed into the following standardized representation of a liquid handle operation. The transformation process is handled by custom Python code that retrieves the operation logs from the ACS application API and parses these into a generic liquid handling event with a list of transfers in it:

```JavaScript
{
    "run_id": "<UUID>",
    "event_id": "111111",
    "username": "ginkgo_user",
    "transfers": [
      {
        "id": null,
        "input": {
          "row": '2',
          "column": '2',
          "container_id": '123456'
        },
        "output": {
          "row": '1',
          "column": '1',
          "container_id": '234567'
        },
        "context": null,
        "lims_run_id": null,
        "liquid_class": null,
        "failure_reason": null,
        "actual_volume_uL": 5,
        "lims_transfer_id": null,
        "requested_volume_uL": 5,
        "occurred_at_iso_timestamp": "2024-09-09T22:42:03.257853+00:00"
      },
      {
        "id": null,
        "input": {
          "row": '4',
          "column": '2',
          "container_id": '123456'
        },
        "output": {
          "row": '2',
          "column": '1',
          "container_id": '234567'
        },
        "context": null,
        "lims_run_id": null,
        "liquid_class": null,
        "failure_reason": null,
        "actual_volume_uL": '5',
        "lims_transfer_id": null,
        "requested_volume_uL": '5',
        "occurred_at_iso_timestamp": "2024-09-09T22:42:03.257853+00:00"
      }
    ],
    "event_type": "liquid_handled",
    "workcell_id": "nebula",
    "iso_timestamp": "2024-09-09T22:42:03.323335+00:00",
    "operation_type": "stamp",
    "instrument_name": "bravo-96-sm-1",
    "instrument_type": "bravo-96",
    "logged_by_service": {
      "name": "acs-extractor",
      "version": "56"
    }
}
```

The extractor passes this transformed event to the EPP's Event Store, which persists it to a database with a unique generated identifier.

Asynchronously, a separate polling task monitors for newly created events in the Event Store and evaluates dependencies between events to determine which events are available for processing. When an event is identified as ready for processing, it is passed to the matching event processor in EPP for its standardized type. The processor is a Python function that is responsible for any updates that need to be performed in Ginkgo's LIMS and workflow management systems. Critically, this function is written to be idempotent - if it is called multiple times because it fails for any reason, the output remains as expected. Upon completion of its duties, the processor notifies the Event Store that processing has successfully completed.

This paradigm in EPP offers a number of advantages:

- Events from different automation platforms can be processed in the same way
- Introducing the buffer of an intermediate Event Store allows events from independent runs to be processed in parallel, even in the case of processing errors in other runs.
- Additional interfaces and data products can be built on top of the standardized event data. For example, Ginkgo has built a web application that allows users to remediate event processing when errors occur.

While EPP was designed for Ginkgo's internal software platform, it is adaptable to other LIMS and data processing tools due to its use of standard Python functions for event processing tasks. We are considering productizing EPP in simplified form for Ginkgo Automation's customers if there is enough interest. Please let us know if you are interested in a product like this.

Summary

Ginkgo was able to smoothly integrate RACs into its EPP framework because ACS includes modern integration points such as a Kafka-based messaging bus and RESTful APIs. This full integration powers in-order processing of events from various automation platforms with support for remediation in a custom UI. A standardized format for automation events allows for event processing without the need for special handlers for each different type of automation system.

CONCLUSION

The best integration option for a given customer depends on their particular situation. Customers that need lightweight integrations with their other software systems can simply use the tools provided by Ginkgo Automation as in Case Study A. We believe this integration mode will be the most common. Customers with sizable software and hardware internal platforms already in place may wish to build fully integrated solutions that plug ACS into their workflow and data ecosystem, like in Case Studies B and C. At Ginkgo Automation, we understand very well what it takes to integrate the control and data planes in high-throughput scenarios. Our design goal is to provide the best integration flexibility and speed in the industry for this. If you are interested in discussing how to best integrate ACS with your software systems, please reach out to the Ginkgo Automation team today. ↗