Universal Routing Reference

New or Updated Function Descriptions

4/30/2025

Contents

New or Updated Function Descriptions

The following functions have been either newly added or updated after the *Universal Routing 8.1 Reference Manual* was last published. Unless otherwise noted, these functions are available in IRD's Function object. Other functions are located in IRD's Selection object.

ExcludeAgents

Parameters: Agents: STRING (comma-separated list of agent IDs or variable)
Return value type: STRING

This function instructs URS not to route interactions to any agent on the specified list of agents. Parameter Agents is comma-separated list of agent IDs. Function returns the previous list of excluded agents.

Previous to 7.6, if a target was selected (if it was ready according to Stat Server and reserved) and then excluded from the list of valid targets using the ExcludeAgents function, this target was not actually excluded. Starting with 7.6, the ExcludeAgents function does exclude the agent in the above scenario.

Note: When URS executes the ExcludeAgents function for an interaction, the URS-provided list of excluded agents will be applied to the current or any future Selection objects. The effect of the ExcludeAgents execution can be cancelled only by the execution of another ExcludeAgents function or if URS stops this interaction processing.

Warning! Function ExcludeAgents affects IVR targets.

FindConfigObject

As of 11/24/15, the existing FindConfigObject function is extended to support additional object types as shown in the FindConfigObject Returned Results table below.

Look Up Agent Name, Media logged Into and DN, from Agent Login ID and by Employee ID

Function FindConfigObject with object type CFGPerson and function TargetState can be used to look up an agent name, the media channel logged into, and the agent's DN, based on an agent's Login ID or Employee ID.

Function TargetState[‘EmployeeID.A’] with input parameter agent EmployeeID returns agent readiness information, a list of available medias and DNs.

For example, to use FindConfigObject to find agent (Person) information, the following search criteria can be used:

- DBID of agent: `FindConfigObject[CFGPerson, 'dbid:SomeDBID']`
- DBID of one of agent's logins: `FindConfigObject[CFGPerson, 'logindbid:SomeDBID']`
- EmployeeID of agent: `FindConfigObject[CFGPerson, 'employeeid:SomeEmployeeID']`
- Agents Login: `FindConfigObject[CFGPerson, 'switch:SomeSwitchName|login:SomeLogin']`

FindConfigObject Function Description

Parameters: TYPE: INTEGER
Valid Values: See FindConfigObject Returned Results table
Properties: LIST or variable (provide list of search criteria)
Return value type: LIST

This function returns information about a requested configuration object. You must specify the type of object for which to search (for example, CFGPlace) and the list-presenting search criteria. A valid set of search properties consists of either the name or a combination of the switch and number.
Examples:

- For CFGDN objects, name specifies an alias of the required DN, switch specifies the name of the switch to which the DN must belong, and number specifies this DN number.
- For CFGPlace object, name specifies the name of required place, switch specifies the name of the switch to which a DN (from among the DNs belonging to this Place) belongs, and number specifies the number of this DN.

The search criteria specifies a subset of the object properties, while the function provides the rest of the data. The search criteria must be a unique subset of the properties that identify the configuration object. See the following search criteria and results and also the table below:

Search Criteria: `FindConfigObject[CFGDN, 'name:2201_vit_sw2']` or `FindConfigObject[CFGDN, 'number:2201|switch:vit_sw2']`
Results: `"dbid:1122|name:Place_102_vit_sw2|tenantdbid:103|tenant:Vit|#1.number:102|#1.switch:vit_sw2|#1.type:2|#2.number:112|#2.switch:vit_sw2|#2.type: 1|dns:2".`

When search criteria is not based on dbid, but on names, the tenant is required for an object search. By default, the tenant is the one for the current interaction, but tenant also can be explicitly specified with extra keys: tenant or tenantdbid.

In addition to specifying a single Configuration object, you can also specify a collection of objects. In such cases, the search criteria must contain key all with value true. Additionally, the search criteria might contain extra filters on the Annex (only objects with this value of Annexes are returned). For example, annex.section,option1:value1|annex.section,option2:value2. When specifying a collection of objects, URS returns a reduced set of objects properties, all highlighted in bold, in the table below. Specifying a collection of objects is not supported for tenants, applications, folders, and enumerator values.

FindConfigObject Returned Results

Object Type	Search Key Combinations	Returned Properties
CFGSwitch	dbid,	**dbid**, type, link, **name**, tenant, tenantdbid, **tserverdbid**,

Object Type	Search Key Combinations	Returned Properties
	name	**tserver**, folders, annex, targetdata
CFGFDN	dbid, name switch or switchdbid+number	**dbid, type, number, name, switchdbid, switch**, tenant, tenantdbid, annex, targetdata
CFGPlace	dbid, name switch or switchdbid+number	**dbid, name**, tenant, tenantdbid, annex, targetdata, dns, folder, annex
CFGPerson	dbid, logindbid employeeid switch or switchdbid+login	**dbid, employeeid, firstname, lastname, username, email**, externalid, tenantdbid, tenant, **placedbid, skills**, logins, folders, annex
CFGTenant	dbid, name	dbid, name, annex,
CFGApplication	dbid, name switch or switchdbid	dbid, type, name, workdir, commandline, hostname, hostip, port, switch, servers, annex
CFGSkill	dbid, name	**dbid, name**, tenant, tenantdbid, annex
CFGAgentLogin	dbid, name switch or switchdbid+login	**dbid, login, switchdbid**, switch, tenant, tenantdbid, override, annex
CFGTransaction	dbid, type+name	**dbid, type, name**, tenant, tenantdbid, alias, description, annex
CFGStatDay	dbid, name	**dbid, name**, tenant, tenantdbid, dayofweek, day, starttime, endtime, min, max, target, rate, annex
CFGFolder	dbid	dbid, name, type, class, ownertype, ownerdbid, size, folders, annex
CFGEnumerator	dbid name	**dbid, type, name**, tenant, tenantdbid, description, **displayname**, annex
CFGEnumeratorValue	dbid enumeratordbid+name enumeratorname+name	dbid, enumeratordbid, name, description, displayname, isdefault, annex

GetInteractionAge

Parameters: None
Return value type: FLOAT

This function returns the time difference in seconds (with milliseconds precision) between the current moment in time and the age of the interaction timestamp.

PriorityTuning

Update the Warning on page 606 of the *Universal Routing 8.1 Reference Manual* as follows:

Warning! The interaction selection criteria associated with the `PriorityTuning` function (age of interaction, relative wait time (such as wait time in queue or predictive wait time), service objective risk factor, or any combination of these parameters) are only supported in a multi-URS environments where the same target might be selected by different instances of URSs if:

- all URS instances have the same value of option `use_service_objective` and
- all strategies running/served by URSes include the `PriorityTuning` function with the same parameters values across all strategies.

RequestRouter

See the `RequestRouter` function in Estimated Waiting Time Improvement.

run

Starting with release 8.1.400.39, Universal Routing adds support for the `run` function in skill and threshold expressions.

Parameters:

- for threshold expression:
  subroutine: STRING (Subroutine Name)

  param1: STRING

  param2: STRING
- for skill expression:
  subroutine: STRING (Subroutine Name)

  Agent: will be provided to subroutine by URS

  Virtual Queue: will be provided to subroutine by URS

param1: STRING

param2: STRING

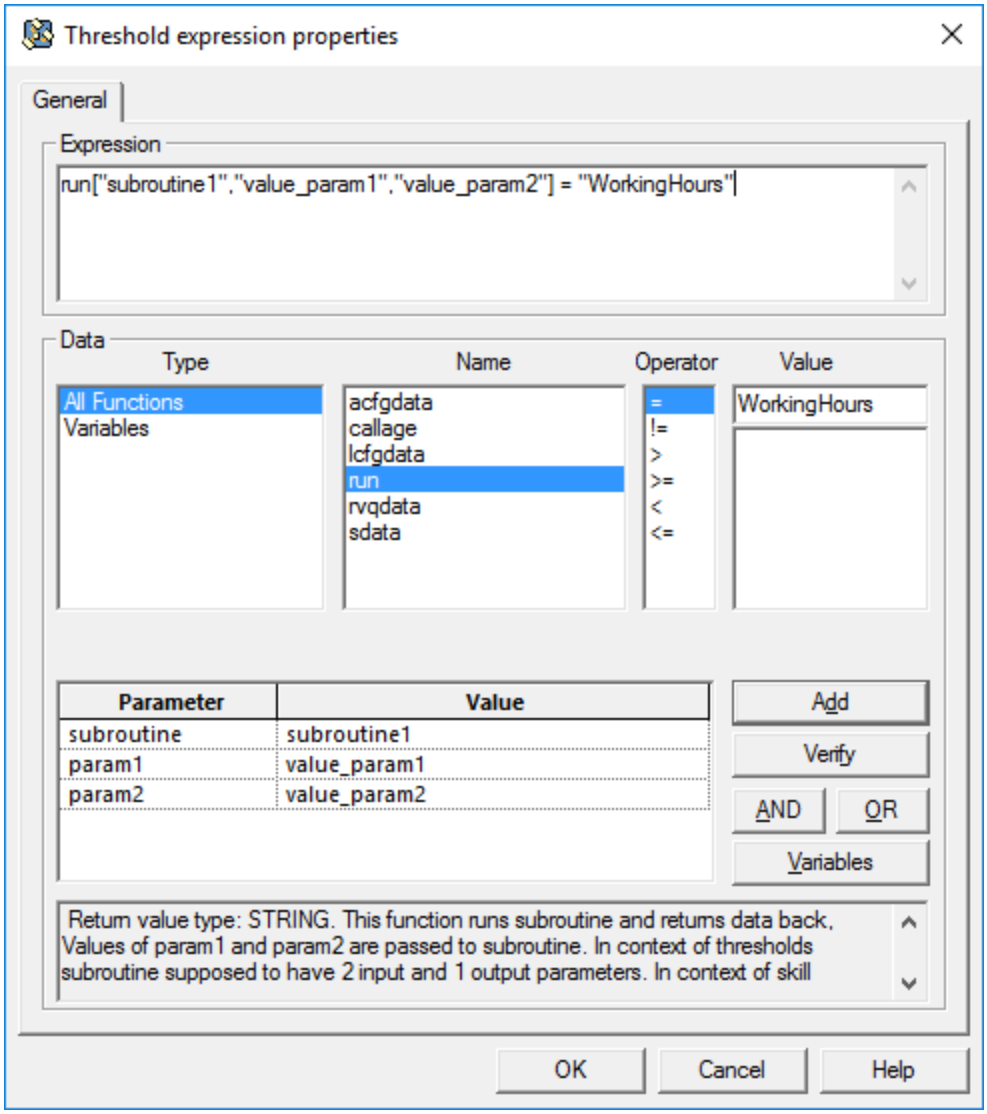
Return value type: STRING

This function executes a defined subroutine with provided parameters. The values of all parameters are passed to the subroutine as input parameters. The subroutine returns 1 output parameter value. The `run` function returns a STRING data type. If the expression is comparing a returned value with some other value, the comparison may not work. If needed, returned data can be explicitly converted to a number via the type-converting functions `num` or `int`.

When accessing the function from a threshold expression, IRD shows only subroutines with 2 input and 1 output parameters.

For example:

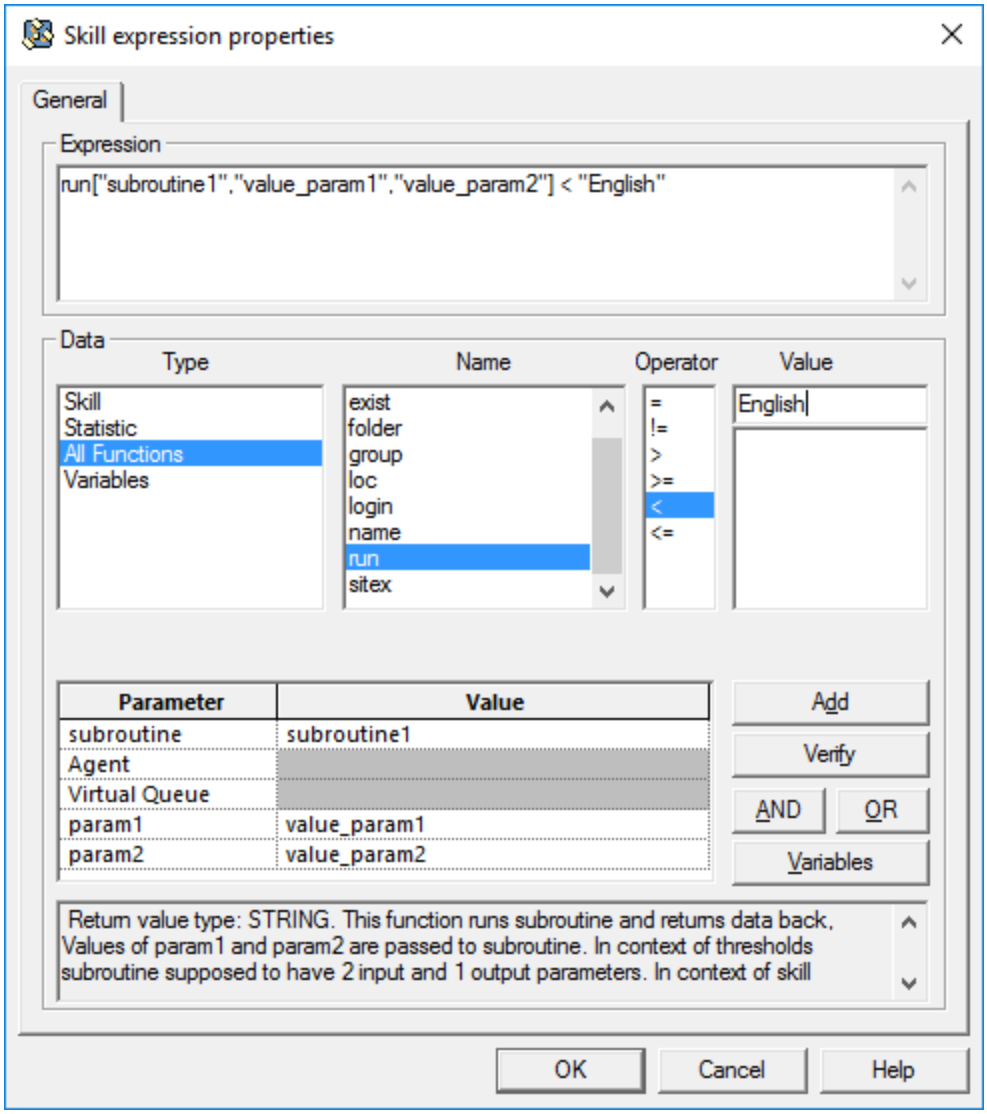
```
num[run[“subroutine1”,”value_param1”, ”value_param2”]] = “5”

run[“subroutine1”,”value_param1”, ”value_param2”] = “WorkingHours”
```



When accessing function from skill expression, IRD shows only subroutines that have 4 input and 1 output parameters.

For example:

```
run[“subroutine1”,”value_param1”, ”value_param2”]<”English”
```



Limitations:

A subroutine called by the run function cannot execute any waiting function, access external servers for some data, or invoke another subroutine. Any attempt to do so raises an error and terminates execution of skill or threshold expressions. All data that the subroutine is allowed to use must reside in URS memory, which means the subroutine can use data only from Configuration Server, Stat Server, interaction data, and data stored in URS memory.

SetIdealAgent

See the `SetIdealAgent` function in Using Agent Skills for Agents/Calls Prioritization.

StrAsciiTok

On page 583 of the *Universal Routing 8.1 Reference Manual*, the description of the string manipulation function StrAsciiTok should read "Every time the function StrAsciiTok is called, it stores in memory the index in String of the ***next*** character of the obtained substring." The manual incorrectly states "...it stores in memory the index in String of the ***last*** character...."

TargetListSelected

See the `TargetListSelected` function in Using Agent Skills for Agents/Calls Prioritization.

TargetState

Update `ready:1` in the TargetState function description, results—LIST Fields as follows:

`ready:1` The agent is considered (by URS) to be ready. If agent capacity is not used, then URS flags the agent as ready if his state is reported by Stat Server as `WaitForNextCall`. If agent capacity is used then the agent has no state, but only a list of medias, which results in URS flagging the agent as ready if this list of medias is not empty. By default, URS takes the ready flag into account when selecting agents, but it can be overwritten in the strategy by calling function `CheckAgentState` with an argument of `false`. This will cause URS to ignore the ready flag when looking for an available agent for the current call.

You can use the `FindConfigObject` and `TargetState` functions look up an agent name, the media channel logged into, and the agent's DN, based on an agent's Login ID or Employee ID.

Function `TargetState['EmplooyeeID.A']` returns agent readiness information, a list of available medias and DNs. This information is in addition to the returned results listed in the function description in the *Universal Routing 8.1 Reference Manual*.

Also see FindConfigObject.

transfer-to-agent

Location in Configuration Layer by precedence: T-Server, URS
Default value: `false`
Valid values: `true`, `false`,`never`
Value changes: take effect immediately

Instructs URS to request T-Server to transfer interactions from an IVR directly to a target agent instead of returning them to the routing point.

- `true` - allow transferring from IVR to agent.
- `false` - do not allow transferring from IVR to agent, except cases when IVR is Routing Point itself.

- never - do not allow transferring from IVR to agent in all cases.

Use this function where standard routing scenarios do not apply or cannot be used to override the usual method of routing a call to an agent. For example, there may be special hardware or reporting needs. Allows you to initiate a direct transfer to an agent using T-Library functionality. For more information on T-Library functions,see the *TLib Reference Guide*.

Selection Object Functions

To access these functions in IRD's Routing Design view, click **Routing** > **Selection** to open the the Selection properties dialog box. Cilck **Add Item**. Under **Type**, select **Skill.** Under **Name**, select **All Functions**.

cfgdata

Parameters:

- `folder: STRING` (Section name on the Annex tab of the `Person` object)
- `option: STRING` (Option name in the section on the Annex tab of the `Person` object)
- `default: FLOAT` (Preset value is returned, if the option is not found)

Return value type: `FLOAT`

This function returns a numeric value. If the agent's option is found, it returns `1`. If the option is not found and the default value is not specified, it returns `0`. If the option is not found and the default value is specified, the default value is returned.

exist

Parameters: `Skill Name: STRING` (Skill name)
Return value type: `FLOAT`

This function checks if an agent has the provided skill and is applied directly to skill names. It returns `1`, if an agent has the skill, or `0` otherwise.

folder

Parameters: `template: STRING` (Folder name in the Configuration Layer under Persons)
Return value type: `FLOAT`

This function checks if an agent is configured in a folder with a name that matches the specified template. It returns `1`, if there is a match, or `0`, otherwise.

group

Parameters: `Agent Group: STRING` (Agent Group name)
Return value type: `FLOAT`

This function checks if an agent belongs to the specified group. It returns 1, if yes, or 0, otherwise.

loc

Parameters: `Switch Name: STRING` (Switch name)
Return value type: `FLOAT`

This function verifies the agent location. It returns 1, if an agent has a DN belonging to the specified Switch, or 0, otherwise.

login

Parameters: `media: STRING` (Media name)
Return value type: `FLOAT`

This function returns 1, if an agent is logged in to provided media, or 0, otherwise. If media is set to any and an agent is logged in to at least one media, then 1 is returned.

name

Parameters: `template: STRING` (Agent employee ID template)
Return value type: `FLOAT`

This function checks if an agent name matches the specified template. It returns 1, if there is a match, or 0, otherwise.

sitex

Return value type: `FLOAT`

The `sitex` function verifies an agent's location (site). It returns 1 if the agent was logged in at the provided site and 0 otherwise. If the name of the site is set to `this`, then instead of checking, the function just returns the site name. The agent's site is usually provided by the agent himself through the `site` parameter in the `Reasons` attribute of the agent ready request. **Note:** In SIP cluster configuration, it is automatically populated by SIP Server.

String Manipulation Functions

The following string manipulation functions are available in IRD starting with version 8.1.400.39:

StrUTF8Encode

Parameters: `String: STRING` or variable (representing the string)
Return value type: `STRING`

This function performs encoding of the provided multibyte String into UTF8 format. Encoding is done based on current locale.

StrUTF8Decode

Parameters: `String: STRING` or variable (representing the string)
Return value type: `STRING`

This function performs decoding of the provided UTF8 String into multibyte format. Decoding is done based on current locale.

StrURLEncode

Parameters:

- `String: STRING` or variable (representing the string)
- `UTF8: INTEGER` or variable (representing the integer true/false value)

Return value type: `STRING`

This function performs URL Encoding of the input String. Additionally UTF8 conversion might also be applied before URL encoding if the UTF8 parameter is true (on any integer that is not 0). If UTF8 encoding is specified but fails, then URL encoding will be applied to the original string.

Note: URL encoding is the process of replacing unprintable characters within URLs, such as -, _, ., !, ~, *, ', (, and), with their corresponding hexadecimal code prefixed with %. The URL encoded value is safe to use as a value for the Web URL parameters.

StrURLDecode

Parameters:

- `String: STRING` or variable (representing the string)
- `UTF8: INTEGER` or variable (representing the integer true/false value)

Return value type: `STRING`

This function performs URL decoding from the input string. Additionally UTF8 conversion might also be applied after URL decoding if the UTF8 parameter is true (on any integer that is not 0). If UTF8 decoding is specified but fails, then just the result of the URL decoding process will be returned.

Note: URL decoding is the process of replacing the URL encoded characters with their corresponding original characters and replacing the + sign with a space.

Support for JavaScript

Beginning with release 8.1.400.67, URS supports execution of stand-alone JavaScript strategies and subroutines. The supported standard is **ECMA-327**, 3rd edition.

IRD now provides a possibility to write a part of or the entire strategy logic using the JavaScript

language in Macro or Script objects.

- To enable Script objects, navigate to the **Tools/Routing Design Options** dialog and in the **Views** tab select the **Routing Design/Scripts** checkbox.
- For Script objects, use the **Script/ecma** type.
- For Macro objects, select the **Complex macro** checkbox.
- Write the script source code in the **Definition** tab and always use the **Verify** button to check the validity of the script or macro you create.

The objects you create can be used in regular IRD strategies/subroutines. Script objects can also be used on their own.

To pass data from the calling strategy to the Script object:

Accessing of input parameters requires writing explicit code that will do it. Calling the strategy will put input parameters in a stack. In scripts, use the `GetInputParams()` function to retrieve the input parameters. Retrieved data will be removed from the stack, so calling the function more than once in a Script will result in an error. It is recommended to call the `GetInputParams()` function at the very beginning of the script.

To pass data from the Script object to the calling strategy:

Use the `SetOutputParams(object)` function to return output parameters.

> ### Important
> If the script is used as a subroutine, then the last executed command should be a Return(error_code); where error code is **0** if all OK and a non-zero error code if any error occurs. The following is a sample script with 2 input parameters (**i1** and **i2**) and 2 output parameters (**out1** and **out2**):
>
> ```
> var inp = GetInputParams();
> var res= new Object();
> res.out1= inp.i1+inp.i2;
> res.out2= inp.i1-inp.i2;
> SetOutputParams(res);
> Return(0);
> ```

To pass data to a Macro:

Macros are called as subroutines. All macro parameters are considered as input and passed automatically, there are no output params.

To pass data from a Macro to the calling strategy:

Data can be passed back to the calling strategy from Macro through the following:

- stack – combination of `Push(data)` and `data=Pop()`functions.
- wake up calls – combination of `WakeCallUp('', data)` and `data=WakeUpData['']` functions.

- dedicated returning – combination of `ReturnEx(error, data)` and data=`ReturnData[]`functions.

INTERACTION scope variables can be used anywhere (Scripts, Macros, strategies, subroutines) to pass data in any direction.

URS JavaScript Code/Functions Support:

- Standard **ECMA-327**, ECMAScript 3rd Edition Compact Profile.
- All operators must be ended with ";".
- Internal characters' presentation – multibyte (not UTF16).

Functions for JavaScript Strategies

All URS functions can be used. Functions are listed in the **compiler.dat** file and located in the IRD installation directory. The following functions have mostly been created for use in JavaScript strategies (though they also can be used in IRD strategies, if needed):

GetCallObject()

The `GetCallObject()` function returns an object presenting the current call. The properties of this object can be used to access (read and sometimes write) real call data. This call object has the following properties:

- **udata** or **userdata** – allows read/write access to call attached data
  - `GetCallObject().udata.abc = 123`; the same as `Update('{d}abc', 123)`
  - `X = GetCallObject().udata.abc`; the same as `X = UData ('abc')`
  - `X = GetCallObject().udata['*']`; returns all calls user data
  - `GetCallObject().udata.abc = null`; the same as `DeleteAttachedData ('abc')`
  - `GetCallObject().udata.abc = undefined`; the same as `DeleteAttachedData ('abc')`
  - `delete GetCallObject().udata.abc`; the same as `DeleteAttachedData ('abc')`
  - `GetCallObject().udata.abc+ = 5`; take current value of attached data, increment it by 5 and re-attach.
  - `GetCallObject().udata+ = {"abc":123, "def":"aaa"}`; the same as `Update('', '{d}abc:123|def:aaa')`
  - `GetCallObject().udata- = "abc"`; the same as `DeleteAttachedData ('abc')`
- **xdata** or **extensions** – allows read/write access to call extensions data
- **voice** – allows read access to following call data: `thisdn` or `_dest`, `_orig`, `acdqueue`, `type`, `ani`, `dnis` or `contactedaddr`, `ced`, `media` or `category`, `customerid`, `connid`, `g_uid`, `callid`, `trunk`, `media_server`, and `control_server`
- **location** – allows read access to following call data: `media_server` and `control_server`
- **data** – allows read/write access to the call's `INTERACTION` scope variables

The following functions for target selection are preferred for performing target selection from JavaScript strategies:

PutIntoQueue(id, virtual queue, priority, statistic, selection flag, target, . . .)

This function is similar to the SelectDN function with following differences:

- It allows explicit control of call queues with the first parameter. In the `SelectDN` function, `id` is selected by the compiler and they are all different within a strategy. Using the same `id` in different instances of the `PutIntoQueue` function will result in the call's requeueing (call targeting resulted from an older `PutIntoQueue` function will be eliminated and replaced with the new one). If `id` is set to **0** then the new `id`-less call's queue will always be created.
- It only puts the call into queue but does not try to select a target. The function returns the `refid` that can later be used for target selection.

RemoveFromQueue(id, virtual queue)

This function is similar to the ClearTargets function, but allows more explicit control over excluding a call from one or another queue.

- If the `virtual queue` is empty and `id` is not **0**, then the call will be removed only from the call queue in which it was placed by the `PutIntoQueue` function using the same `id`.
- If the `virtual queue` is empty and the `id` is **0**, then the call will be removed from all call queues as well as from all virtual queues (Virtual queue EventDiverted(redirected) will be distributed for all virtual queues call is in).
- If virtual queue is not empty the n call will be removed from the all call's queues associated with provided virtual queue as well as from this virtual queue (the virtual queue `EventDiverted`(redirected) event will be distributed for this virtual queue).

RemoveFromQueueN(refid)

This function is similar to the RemoveFromQueue function, but it removes the call from call queue identified by `refid` returned by the `PutIntoQueue` function. The call is not removed from any virtual queue.

SelectTargetFromQueue(refid, timeout)

This function is similar to the `SuspendForDN` function (and in case of *refid=0*, they are identical). The function places the call into a *waiting for targets from all queues calls are in* state for the duration of the provided timeout. Before doing that function explicitly tries to select target (by statistics) from the queue the call is in, pointed by the provided `refid` (return value of the `PutIntoQueue` function). If `refid` is **0**, then all call queues will be tried. The returned value is identical to the value provided by the SuspendForDN function.

RouteToTarget(target)

This function is a more advanced version of the `RouteCall` function. Similar to the `RouteCall` function it accepts a target returned by the `SuspendForDN` function or the `SelectTargetFromQueue` function and routes the call to the provided target. The returned value is identical to the value returned by the `RouteCall` function.

The main difference is that this function will try all means to route the call; that is, answer the call if

needed, return it back to the original Routing Point, and if the call is on an uninterruptible treatment or in a transition state, it will wait until routing is possible again. It is this function, and not the `RouteCall` function that IRD uses when custom routing is activated in Routing objects.

JavaScript support for Treatment functions

The function, `TreatmentPlayAnnouncement`, and other `Treatment<TreatmentType>` functions which are used to start and wait for treatment ending can not be used in JavaScript directly due to a name conflict with the same name constant.

The recommended way to use this function is by referring to it indirectly through a URS functional module. For example:

```
var funcTreatmentPlayAnouncement= FunctionalModule('[www.genesyslab.com/modules/
urs')['TreatmentPlayAnnouncement'];
funcTreatmentPlayAnouncement('LANGUAGE', 'English(US)', 'PROMPT.1.TEXT', 'http://127.1.1.1/
audios/bienvenida_001.wav');
```

Alternatively the function, `StartTreatmentPlayAnnoncement` (followed with `SuspendForTreatmentEnd`) can be used instead in this case. Effectively, the `Treatment<TreatmentType>` functions can be considered as shortcuts for combinations of two functions `StartTreatment<TreatmentType>` and `SuspendForTreatmentEnd`.

Notice how parameters in the above sample are passed to `Treatment<TreatmentType>` or `StartTreatment<TreatmentType>` functions, it is different from passing them to busy treatment functions. The parameters must be passed directly and as a sequence of keys and values: key, value, key, value, and so on.

Sample JavaScript Snippet

The following is a sample JS snippet where the call is routed to any available agent during working hours and if outside working hours, the not the strategy plays an appropriate treatment. The treatments to play are obtained from some a Web Service.

```
var funcTreatmentPlayApplication= FunctionalModule('[www.genesyslab.com/modules/
urs')['TreatmentPlayApplication'];

function RouteAnyAgent(greeting, busy)
{
   funcTreatmentPlayApplication('APP_ID',greeting);
   var queue= PutIntoQueue(1, '', 0, "StatAgentLoading", SelectMin, '?:2>1.GA');
   AddBusyTreatment(TreatmentPlayApplication, 'APP_ID:' + busy, 0);
   var target= SelectTargetFromQueue(queue, 60);
   if (!Failed())
     RouteToTarget(target);
}

function RoutetoAnyAgentWorkingHours(greeting, weekend, outoftime, busy)
{
    if (Date.getDay()==0 || Date.getDay()==6)
       funcTreatmentPlayApplication('APP_ID',weekend);
    else if (Date.getHours()<8 || Date.getHours()>18)
       funcTreatmentPlayApplication('APP_ID',outoftime);
    else
       RouteAnyAgent(greeting, busy);
}
```

```
x= ObjectFromJSON(
       GetHttpRequestInfo(1,
            StrFormat("http://myhost:myport/MyMessage?ani=~s&dnis=~s&connid=~s",
            ANI(), DNIS(), ConnID()), "", "", 0, "", ""));

if (!Failed())
      RoutetoAnyAgentWorkingHours(x.greeting, x.weekend , x.outoftime , x.busy);

Default();
```

Secure Connections Support Includes SNI Functionality

Starting with release 8.1.400.71, URS supports Server Name Indication (SNI) extension for TLS handshakes. As a result, HTTPS resources can be contacted from within URS, that is, using the Web Service block in IRD.

If the SNI functionality is activated, URS adds an extra parameter, `tls-target-name`, into the transport parameters of the connecting requests, set to the name of the host the web request is directed to.

The SNI functionality can be activated using one of the following methods:

- Set the `def_sni` option to `true`. For a description of the new option, refer to the New or Updated Option Descriptions topic.
- Specify `sni:true` in the **Hints** field of the IRD Web Service object. The **Hints** field is located under **TLS group** in the **Security** tab of the Web Server object.

> ### Important
> URS first uses the value from the IRD Web Service object. If that is absent, it uses the value set by the `def_sni` URS option.

Optimal Skill Update Mode

Starting with release 8.1.400.75, URS is enhanced to provide a possibility for more optimally updating skill expressions when part of the skill expression uses the `login` function. The optimal skill update mode is activated when the new URS option, `content_update_on_login`, is set to `false`. Setting it to `force`, which is the default value, overrides the enhancement and retains the behavior from the previous URS versions.

`content_update_on_login`
Location: `default` section of URS `Application` object
Default Value: `force`
Valid Values: `force`, `false`
Changes Take Effect: Immediately

Timeout to Wait before Sending Negative Response to Web Client

Beginning with version 8.1.400.78, URS allows users to specify a timeout to wait for before sending a negative response to a web client's request to perform an operation for an interaction that URS does not have. Timeout (in seconds) can be defined in the additional parameter, **maxdelay**, in the web request to URS or in the **call_sync_time** option.

call_sync_time
Location: `http` section of URS `Application` object
Default Value: `0`
Valid Values: `0` to `10`
Changes Take Effect: After restart

The value defined in the **call_sync_time** option is used if the **maxdelay** parameter is not specified.

For example, `urs/call/ConnID/func?name=Timeout¶ms=[120]&maxdelay=2`. Here, URS is requested to apply the `Timeout[120]` function to the interaction. If the requested interaction is not found, then URS will wait up to 2 seconds. If URS gets the requested interaction during the 2 seconds, it will apply the function to it. Else, it answers with an error.

> **Important**
>
> The maximum value allowed for the `maxdelay` parameter (and the `http/call_sync_time` option) is **10** seconds.

Multithreading Capability

Beginning with release 8.1.400.78, URS is enhanced with a multithreading capability to find matching agents who satisfy the conditions of the specified skill expression, in a given configuration . This improves URS performance in larger environments characterized by agent headcounts exceeding 10,000 or even 100,000 across locations.

A new configuration option, `mts`, is introduced to control the multithreading capability.

mts
Location: `default` section of URS `Application` object
Default Value: `0:0`
Valid Values: `0:0`, `1:0`, `1:1`
Changes Take Effect: Immediately

`0:0` - indicates multithreading is switched off.
`1:0` - indicates multithreading is turned on, but will be applied to expressions containing only skills. Skill expressions with statistics and functions are excluded in single threading mode.

1:1 - multithreading is turned on and skill expressions with statistics and functions are also included for multithreaded processing.

A new console command, `mtskill`, is also provided for exploring the multithreading capability.

Format: `mtskill` <TenantName> <SkillExpression>.

As an output, URS provides 2 corresponding time intervals (in microseconds).

The following **limitations** are to be considered before turning on multithreading:

- Multithreading is justifiable only in very big environments.
- URS must run on very powerful hardware with multiple processors available for URS (running multithreading on single processor machine will slowdown URS).
- URS logging is disabled in multithreaded mode, while URS is updating skill expressions.
- Some skill expression functions, such as `run`, `group`, `folder`, and `tag`, have too big a footprint to be safely used in a multithreaded environment. Skill expressions containing these functions will always be executed in the single threaded mode irrespective of the value of the `mts` option.

Improved EWT Accuracy

EWT accuracy is determined by the difference between the EWT value given when a call enters a queue and the actual wait time that the particular call spent in the queue. When the difference is minimal it translates to a better EWT accuracy. EWT accuracy is impacted if calls of different types are placed into the same VQ. For example, if one VQ is used for both inbound, virtual callback (CB), and CB outbound calls, then EWT accuracy is low because outbound CB calls have some unique properties, such as:

- outbound CB calls are placed in VQ only for a short time of several seconds.
- outbound and virtual CB calls are two call representing the same interaction in a VQ. This creates double counting.
- outbound and virtual CB calls leave the queue at the same time breaking the EWT calculation model, which assumes that one call is distributed at a time and calls are distributed at a constant rate.

EWT accuracy can be increased if outbound CB calls are not taken into account for EWT calculation.

Beginning with release 8.1.400.83, URS provides more accurate EWT calculations for a Virtual Queue when interactions that are not intended to be routed by URS end up in the Virtual Queue. URS now provides the ability to mark and ignore such calls in EWT calculations. You can mark an interaction by setting its run time mode to 524288 and exclude them from EWT calculations.

To mark an interaction you can, use the `SetRunTimeMode` function within the IRD strategy. Or, it is set automatically if `EventRouteRequest` starting a strategy contains `AttributeCallType` with value **3** (outbound) and `AttributeUserData` has the `_CB_N_CALLBACK_ACCEPTED` key set to **1**.

Ignoring such marked calls in EWT calculations is controlled with the new configuration option, `lvq_ignore_duplicates`.

`lvq_ignore_duplicates`
Location: `default` section of URS `Application` object; can also be defined at the VQ level
Default Value: `false`
Valid Values: `true`, `false`
Changes Take Effect: Immediately

Setting the option to `true` ignores marked calls in EWT calculations.

If the option is set to `true` for a VQ and when an interaction marked as 524288 leaves the VQ:

- it will not change the distributing quitting rate from that VQ (its quitting will not go into an array of the last 32 quits). This improves the accuracy of `lvq` requests when `aqt=urs`.
- it will be skipped (not counted) when URS goes through (counts) all calls in the VQ to answer an `lvq` web request. This improves the accuracy of all types of `lvq` requests (`aqt=urs`, `urs2` or `stat`).

If the option is set to `false` for a VQ, then all interactions in that VQ (regulars or duplicates) will be counted for EWT calculations.

Log messages about sending VQ events (such as, *Queued*, *Diverted*) are now extended to indicate if URS counts or ignores the interaction in EWT calculations.

```
12:38:36.134_T_I_00820324eb06cefd \[14:02\] sending event 58 for vq
RBWM_UKFD_Ingress_VQ_EMEA (0 53-7 (0, i=349923 o=349904) 1633001911 170/32) (x)
```
(The x in the above sample message indicates if the interaction is considered by URS as a duplicate. If x is 1, it indicates that the interaction is a duplicate, and if x is 0, it indicates that the interaction is not a duplicate.)

Improved EWT Consistency

Beginning with version 8.1.400.84, URS is enhanced to improve consistency in EWT calculations for web requests that are interactionless and utilize average handling time provided by the default `lvq` URS method.

Improved consistency in EWT calculations is characterised by:

- absence of sharp changes (peaks or drops) in EWT values provided by URS.
- similarity in values of EWT returned by different URS instances in the same environment.

Default behavior

When obtaining EWT values for virtual queues (in response to the `lvq` web request) URS utilizes its own data, that is, calls that it places into virtual queues, targets that these calls are waiting for, and so on. As a result:

- it is possible that there is a sharp change in the provided EWT values if URS switches between sets of data used to calculate EWT.

- values returned by different URS instances might not be the same (if the data used by the different URS instances are also different).

To simulate a global queue perception in a distributed call center, multiple URS instances in the same environment must be able to provide consistent (ideally, the same) EWT value for the same VQ. A caller must get the same EWT value regardless of which data center the call lands in.

URS has the following two data sources for calculating the average handling time (time per call from the VQ) for interactions that are not defined in a web request (`max` is used in place of `connid`):

1. Internal queues created by URS when executing strategies.
2. One of the skill expressions from the internal queues that URS remembers (referred as the VQ Presenter). Internal queues are short lived objects that URS might dispose if not used. It is possible that all internal queues associated with a VQ are deleted. To be able to provide reasonable data even in such cases, URS remembers one of the skill expressions. URS selects the skill expression to remember based on which expression returns the highest number of agents. The identified skill expression represents the VQ when there are no internal queues and is referred to as the **VQ Presenter**.

Usually, URS tries to get data from the internal queues first and if at least one internal queue associated with the VQ exists, then, as listed above, data source 1 is used. If there are no internal queues, data source 2, that is, data provided by the VQ Presenter is used. A side effect to this approach is that URS can spontaneously switch between the two data sources (for interactionless web requests). This might result in the returned EWT values fluctuating frequently.

New enhanced behavior

In the new enhanced behaviour, URS uses the VQ Presenters as the primary source of data for calculating EWT even if internal queues exist.

To facilitate the new behaviour, URS ensures that:

- A VQ Presenter always exists.
  - Any skill expression that the strategy uses can be used as a VQ presenter, even if the skill expression contains statistics.
  - A skill expression identified as a VQ Presenter will not be deleted even if it has not been used for a long time.
- A VQ Presenter can be set or changed only in the following cases:
  - On VQ creation. If a VQ associated with an Agent Group as its origination DN, then this Agent Group will be set as the initial presenter for the VQ.
  - On creating a new internal queue associated with the VQ. At each such instance, sizes of the current VQ presenter and skill expression used by the internal queue are compared and the biggest one is selected as the new VQ presenter.
  - On placing any interaction into the VQ. Re-skilling of agents theoretically might result in reducing the size of the selected presenter and it will no longer be the biggest presenter. As a result, URS constantly rechecks the size of the current VQ presenters whenever a call is added into the VQ. The size of the current presenter and skill expression used by the call to enter the VQ are compared and the biggest one is selected.

Each time a VQ Presenter is changed, an entry is logged. For example, `15:27:23.757_M_I_ [10:85]`

`LVQ NameOfVQ presenter set: <?Agents5_10:>(21499ec7bb0) media=0.`

If different URS instances are executing the same set of strategies, it is likely that all those URS instances will also have the same VQ Presenters.

By deafult, the existing `lvq` method with `aqt=urs2` will continue to work as before. A new configuration option, `lvq_force_presenter`, is introduced to activate the new behaviour.

`lvq_force_presenter`
Location: `default` section of URS `Application` object; can also be defined at the VQ level with section name as `URS application name` or `__ROUTER__`
Default Value: `false`
Valid Values: `true`, `false`
Changes Take Effect: Immediately

Setting the option to `true` activates the new behaviour where a VQ presenter is used as a primary source of information to obtain the average handling time per call.

Reporting

To allow evaluation of the quality of the EWT calculations, URS can be enabled to collect (and report on) data about the estimated and actual waiting times for calls in a virtual queue.

- Every time an interaction enters a virtual queue the current EWTs are obtained and stored inside the interaction.
- Every time an interaction leaves the virtual queue the stored EWTs along with the actual time the interaction was waiting for is stored in the virtual queue. The virtual queue store information only about the latest interaction that quit the queue.

The `lvq` web request is extended to include this information as well as other statistical data that can be useful for tracing processing of calls in one or another virtual queue.

You can use the following requests to query data:

- `urs/call/max/lvq?tenant=TenantName&name=VirtualQueueName&filter=presenter`

- returns the skill expression/agent group used as the current presenter for the specified virtual queue.

- `urs/call/max/lvq?tenant=TenantName&name=VirtualQueueName&filter=trace&ewttrace[=N]`

- `filter=trace` returns tracing data for the specified VQ.
- `ewttrace` or `ewttrace=N` triggers the tracing mode for the specified VQ for the next `N` minutes (by default `N=3`).

For a virtual queue in tracing mode, URS collects extra data about the virtual queue (as permanent collection of such data takes a toll on performance). Additionally, for a VQ in tracing mode, URS records extra information in the log entries even if the `default/verbose` option is set to `false`.

When `filter=trace`, the following data is returned (note that when a value is unknown the field might not be returned):

Field	Description
time	current UTC time
lcalls_in	Local number of calls that have entered the VQ so far.
lcalls_out	Local number of calls that have exited the VQ so far.
lcalls	Local number of calls in the VQ (lcalls_in - lcalls_out).
rlcalls	Local number of real calls in the VQ (lcalls – duplicates).
calls	Global number of calls in the VQ (effectively StatCallsInQueue as returned by StatServer).
mrs	Multi-URS factor (used to convert local data into global (calls/lcalls).
rcalls	An estimate of the global number of real calls (rlcalls * mrs).
aqt_stat	Time per call according to StatServer (=StatExpectedWaitingTime/StatCallsInQueue).
ewt_stat	Waiting time according to StatServer (=aqt_stat * (rcalls+1)).
aqt_urs	Time per call according to URS or global quitting rate (local quiting rate / mrs).
ewt_urs	Waiting time according to URS (=aqt_urs * (rcalls+1)).
aqt_urs2	Time per call according to URS average handling time (calculated as per URS settings).
ewt_urs2	Waiting time according to URS (=aqt_urs2 * (rcalls+1)).
aqt_ursp	Same as aqt_urs2, but aht is calculated based on presenter.
ewt_ursp	Same as aqt_urs2, but aht is caluclated based on presenter.
xid	connid of latest call distributed into the VQ.
xtm	Latest call entry time into the VQ.
xewt_stat	StatServer based estimate of waiting time for the latest call (at the xtm time).
xewt_urs	URS quit rate based estimate of waiting time for latest call (at the xtm time).
xewt_urs2	URS average handling time based estimate of waiting time for latest call (at the xtm time).
xewt_ursp	URS average handling time for presenter based estimate of waiting time for the latest call (at the xtm time).

For a VQ in tracing mode the log message (logged when the call is distributed from the VQ) is as follows:

`12:36:04.200_M_I_03390320b4c930b0 [14:02] LVQ NameOfLVQ (58,1) ewts: xtm, xwt, xewt_stat, xewt_urs, xewt_urs2, xwt_ursp,` (along with some other data).

Note that if the VQ is not traced the above message might still be logged if the log level is set to 4 or 5, but the message will have no data for `xewt_urs2` and `xwt_ursp`.

You can follow one of the two patterns given below for tracing EWT for a VQ with the provided web requests:

1. Periodically (for example, once per minute) you can send URS the following web request, `urs/call/max/lvq?tenant=TenantName&name=VirtualQueueName&filter=trace&ewttrace`. Collect the output data for a period of time (say, a few hours) and visualize the output (for example, as an Excel spreadsheet).
2. Send the following web request to URS, `urs/call/max/lvq?tenant=TenantName&name=VirtualQueueName&filter=trace&ewttrace=180`. Collect URS logs for the next 180 minutes (3 hours), extract the related log messages from the logs and visualize them.

Limitations

The new behaviour is not necessarily better if compared with the default behaviour where URS relies on internal queues. That depends on how a specific solution has been implemented and how virtual queues are used in the solution. It is expected that the new behaviour will work good in cases of cascaded routing.

Also, URS cannot detect by itself, when the usage of one or another virtual queue changes sharply. For instance, solutions/strategies may start to use completely new skill expressions. When the usage of a virtual queue is changed, URS might still continue to use old virtual queues' presenters (if they happen to be bigger). To address such cases and avoid restarting URS to align virtual queue usage, the `lvqs` console command can be used with an extra optional parameter, `reset_presenter`.

For example: `lvqs TenantName VQName reset_presenter`
- where VQName is name of the virtual queue or *.
- All matched virtual queues will have their presenter updated (their presenters will be reset based on the current internal queues URS has for them).

From the Web API, the `lvqs` console command can be executed as, `urs/console?lvqs TenantName VQName reset_presenter`.