

## VERSUS { NETWORKS }

# Versus SDK Reference

The versus object available inside Python Script and HTTP Request workflow steps.

41 core methods · 34 integration methods · 23 native vendor families

Generated from the live SDK · [versusnetworks.com/docs/workflow-sdk-reference](https://versusnetworks.com/docs/workflow-sdk-reference)

---

The `versus` object – call these from a Python script step.

```
# — Core — versus.<method>
versus.add_snow_work_notes(change_number: str, notes: str) -> None
# Append work notes to an existing CHG or CTASK.
#
# Args:
#   change_number – e.g. 'CHG0001234' or 'CTASK0001234'
#   notes         – text to append to work notes
versus.ai(prompt: str, system: str = '', max_tokens: int = 2048) -> str
# Call the default AI provider with a single prompt.
#
# Shortcut for versus.integrations.ai().complete(prompt).
# Uses the provider marked is_default=True; falls back to
# ANTHROPIC_API_KEY env var when no provider is configured.
#
# Args:
#   prompt      – user message / question
#   system      – optional system prompt
#   max_tokens  – cap on response length (default 2048)
#
# Returns the response text as a string.
# Raises RuntimeError on API failure.
#
# Example:
#   summary = versus.ai(f"Summarise this config diff:\n{diff}")
#   severity = versus.ai(
#       f"Classify as critical/high/medium/low:\n{alert}",
#       system="Reply with one word only.",
#       max_tokens=10,
#   )
versus.available_probes(device_id)
# List the native probe kinds available for a device's vendor, e.g.
# ['health', 'ha', 'stats', 'vs_status', 'pool_status', 'objects', 'certs',
#  'interfaces', 'events']. Empty when the vendor has no native driver.
versus.certificates(expiring_within_days=None, expired=False, common_name=None, limit=200)
# Return a list of certificates from the applications inventory.
#
# Keyword args:
#   expiring_within_days – filter to certs expiring within N days
#   expired              – if True, only expired certs
#   common_name          – case-insensitive substring match
#
# Returns list of dicts:
#   {id, common_name, san, issuer, valid_from, valid_to, key_type}
versus.check_ha_status(device_id)
# Check High Availability state for a managed device.
#
# F5 BIG-IP: show /sys failover field-fmt → ACTIVE | STANDBY | FORCED OFFLINE
# Cisco ASA: show failover state          → Active | Standby Ready
# Fortinet:  get system ha status         → Master | Slave
# Palo Alto: show high-availability state
#
# Args:
#   device_id – Device.id
```

```

# Returns raw CLI output string.

versus.check_port(device_id, target_ip, port, protocol='tcp', timeout=5)
# Test TCP/UDP port connectivity from a managed device using nc (netcat).
#
# Args:
#     device_id - Device.id
#     target_ip - destination IP address
#     port      - destination port number (int)
#     protocol  - 'tcp' or 'udp' (default 'tcp')
#     timeout   - connection timeout in seconds (default 5)
#
# Returns 'open', 'closed', or raw error output.

versus.check_snow_approval(change_number: str) -> tuple
# Check whether a CHG or CTASK is approved / in an active state.
#
# Returns (approved: bool, state_label: str, message: str).
# A CHG is approved when approval='approved' or state is Scheduled/Implement.
# A CTASK is approved when state is Work In Progress, Scheduled, or Closed Complete.
#
# Raises RuntimeError if ServiceNow is not configured.

versus.close_snow_change(change_number: str, close_notes: str, work_notes: str = '') -> None
# Close a ServiceNow Change Request (sets state to Review / Closed).
#
# Args:
#     change_number - CHG or CTASK number, e.g. 'CHG0001234'
#     close_notes   - resolution notes (required by SNOW)
#     work_notes    - optional additional work notes

versus.create_snow_change(short_description: str, description: str, risk: str = '3', impact: str = '3', category: str = 'Network', **kwargs) -> dict
# Create a ServiceNow Normal Change Request.
#
# Args:
#     short_description - title
#     description       - full description
#     risk              - '1'=Critical '2'=High '3'=Medium '4'=Low
#     impact            - same scale as risk
#     category          - defaults to 'Network'
#     assigned_to       - username (kwarg)
#     assignment_group  - group name (kwarg)
#     additional_fields - dict of extra SNOW fields (kwarg)
#
# Returns dict: {number, sys_id, state, short_description}

versus.create_snow_incident(short_description: str, description: str, urgency: str = '2', impact: str = '2', category: str = 'Network', **kwargs) -> dict
# Create a ServiceNow Incident.
#
# Args:
#     short_description - title (max 160 chars)
#     description       - full description
#     urgency           - '1'=Critical '2'=High '3'=Medium '4'=Low
#     impact            - same scale as urgency
#     category          - defaults to 'Network'
#     subcategory       - e.g. 'Security' (kwarg)
#     assigned_to       - username (kwarg)
#     assignment_group  - group name (kwarg)
#
# Returns dict: {number, sys_id, state, short_description}

versus.devices(vendor=None, hostname=None, status=None, device_type=None, limit=200)
# Return a list of managed devices.
#
# Keyword args (all optional, AND-combined):
#     vendor      - e.g. 'f5_bigip', 'netscaler', 'panorama'
#     hostname    - case-insensitive substring match
#     status      - e.g. 'online', 'offline'
#     device_type - e.g. 'adc', 'firewall', 'dns_ipam'
#     limit       - max rows returned (default 200)
#

```

```

# Returns list of dicts:
#     {id, hostname, mgmt_ip, vendor, device_type, status,
#       model, software_version, ha_state}

versus.dns_lookup(device_id, hostname)
# Perform a DNS lookup from a managed device.
#
# Args:
#     device_id – Device.id
#     hostname – hostname to resolve
#
# Returns raw nslookup output showing resolved IPs and nameservers.

versus.dns_resolve(hostname)
# Resolve a PUBLIC hostname to its IPs, as an internet client would (this
# runs from the platform's own internet vantage, not a lab device). Flags
# Cloudflare anycast so you can confirm a name is CF-fronted. Returns
# {hostname, addresses:[...], cdn: 'cloudflare'|None} or {error}.

versus.firewall_rules(device_id=None, name=None, action=None, limit=200)
# Return a list of firewall rules.
#
# Returns list of dicts:
#     {id, name, action, context, source_addresses,
#       destination_addresses, services, device_id}

versus.get_arp_table(device_id)
# Get ARP table entries from a managed device.
#
# Useful for verifying MAC-IP mappings and layer 2 reachability.
#
# Args:
#     device_id – Device.id
#
# Returns raw CLI output string.

versus.get_device(device_id)
# Return a single device dict by ID, or None if not found.

versus.get_device_stats(device_id)
# Get CPU, memory, and throughput statistics for a managed device.
#
# F5 BIG-IP: sys performance all-stats + tmm-info
# Cisco:     show processes cpu sorted
# Arista:    show processes top
# Juniper:   show chassis routing-engine
#
# Args:
#     device_id – Device.id
#
# Returns raw CLI output string.

versus.get_input(key, default=None)
# Read a variable set by a previous workflow step.

versus.get_interfaces(device_id)
# Get interface status and error counters from a managed device.
#
# Args:
#     device_id – Device.id
#
# Returns raw CLI output showing interface names, status, speed, and error counts.

versus.get_logs(device_id, pattern=None, lines=50)
# Get recent system / error log entries from a managed device.
#
# Args:
#     device_id – Device.id
#     pattern – optional grep filter pattern (e.g. 'err', 'crit', 'SNAT')
#     lines – number of recent log lines to retrieve (default 50)
#
# Returns raw log output string.

versus.get_pool_status(device_id, pool_name=None)
# Get pool and pool member health from an ADC.

```

```

#
# F5 BIG-IP: TMSH show /ltm/pool members field-fmt
# NetScaler: SSH show lb vserver
#
# Args:
#     device_id - Device.id
#     pool_name - pool name (None = all pools)
#
# Returns raw CLI output. Parse for UP/DOWN member counts and availability.
versus.get_route_table(device_id, prefix=None)
# Get the routing table from a managed device.
#
# Args:
#     device_id - Device.id
#     prefix - optional IP prefix filter (e.g. '10.0.0.0/8')
#
# Returns raw CLI output string.
versus.get_snow_change_state(change_number: str) -> dict
# Lightweight fetch of just the state/approval for a CHG.
#
# Returns dict: {number, sys_id, state, state_label,
#               approval, approved}
# Raises RuntimeError if ServiceNow is not configured.
versus.get_snow_info(change_number: str) -> dict
# Return a summary dict for a CHG or CTASK.
#
# Returns dict: {number, table, state, approval,
#               short_description, sys_id, priority}
# Raises RuntimeError if ServiceNow is not configured.
versus.get_vs_status(device_id, vs_name=None)
# Get virtual server availability and connection statistics from an ADC.
#
# F5 BIG-IP: TMSH show /ltm/virtual field-fmt
# NetScaler: SSH show cs vserver
# A10: SSH show slb virtual-server
#
# Args:
#     device_id - Device.id
#     vs_name - VS name (None = all virtual servers on the device)
#
# Returns raw CLI output. Parse for availability status and connection counts.
versus.http_get(device_id, path)
# Make an iControl REST GET request to an F5 BIG-IP.
#
# Args:
#     device_id - Device.id of the target BIG-IP
#     path - API path, e.g. '/mgmt/tm/ltm/virtual'
#
# Returns the parsed JSON response as a dict.
versus.http_patch(device_id, path, payload)
# Make an iControl REST PATCH request to an F5 BIG-IP.
#
# Args:
#     device_id - Device.id of the target BIG-IP
#     path - API path
#     payload - dict to send as JSON body
#
# Returns the parsed JSON response.
versus.http_probe(url, method='GET', timeout=15)
# HTTP GET/HEAD a URL from the platform's internet vantage - i.e. exactly
# what an external user hitting a Cloudflare-fronted app (lab.versus-demo.com)
# experiences. Read-only (GET/HEAD only). Returns status, edge headers
# (cf-ray/server/cf-cache-status), latency, and a body snippet. Cloudflare
# origin failures surface as 52x (521 down / 522 timeout / 523 unreachable /
# 526 bad cert) with a cf-ray - the direct signal that CF can't reach the lab.
versus.integrations (property)

```

```

# Access all configured integrations by type and name.
#
# Each method returns a client regardless of whether any instances are
# configured – callers should wrap in try/except for graceful fallback.
#
# SIEM (Splunk, Sentinel, Elastic, QRadar, Wazuh, Sumo Logic, Datadog):
#     versus.integrations.siem_list()                → [{id, name, source_type, status}]
#     versus.integrations.siem('Prod Splunk')        → UnifiedSIEMClient
#         .get_alerts(hours=24, severity=None)        → list
#         .get_alerts_for_device(ip, hours=24)        → list
#         .search(query, hours=24)                   → list
#
# SCM (GitHub, GitLab, Bitbucket, Azure DevOps):
#     versus.integrations.scm_list()                 → [{id, name, scm_type}]
#     versus.integrations.scm('GitHub Corp')         → UnifiedSCMClient
#         .recent_commits(repo, hours=24)            → list
#         .create_issue(repo, title, body)           → dict
#
# vSphere / vCenter:
#     versus.integrations.vsphere_list()             → [{id, name, url, status}]
#     versus.integrations.vsphere('vCenter Prod')    → UnifiedVSphereClient
#         .find_vm(name)                             → list
#         .inventory_summary()                       → dict
#         .get_powered_off_vms()                     → list
#
# IPAM (Infoblox, NetBox, phpIPAM, BlueCat, EfficientIP):
#     versus.integrations.ipam_list()                → [{id, name, source_type}]
#     versus.integrations.ipam()                    → UnifiedIPAMClient
#         .search_host(fqdn)                         → list
#         .subnet_utilization(cidr)                  → dict
#         .find_network(cidr)                       → list
#
# CrowdStrike Falcon:
#     versus.integrations.crowdstrike_list()         → [{id, name, base_url}]
#     versus.integrations.crowdstrike()              → UnifiedCrowdStrikeClient
#         .find_host(hostname)                      → dict | None
#         .list_detections()                        → list
#         .get_host_vulnerabilities(aid)             → list
#         .run_rtr_script(aid, script)               → str
#
# Ansible Tower / AWX:
#     versus.integrations.awx_list()                 → [{id, name, url}]
#     versus.integrations.awx()                     → UnifiedAWXClient
#         .run_and_wait(template_id, extra_vars={}) → dict
#         .get_job_stdout(job_id)                   → str
#
# Rundeck:
#     versus.integrations.rundeck_list()             → [{id, name, url}]
#     versus.integrations.rundeck()                  → UnifiedRundeckClient
#         .run_and_wait(job_name, options={})        → dict
#         .get_output_text(execution_id)            → str
#
# Pager / On-call (PagerDuty, Opsgenie, VictorOps, xMatters):
#     versus.integrations.pager_list()               → [{id, name, vendor}]
#     versus.integrations.pager()                    → UnifiedPagerClient
#         .critical(title, body='')                  → dedup_key str
#         .warning(title, body='')                   → dedup_key str
#
# Slack:
#     versus.integrations.slack_list()               → [{id, name, workspace_name}]
#     versus.integrations.slack()                    → UnifiedSlackClient
#         .alert(text, severity='info', title='')    → dict
#         .message(text, channel=None)               → dict
#
# Jira:
#     versus.integrations.jira_list()                → [{id, name, deployment}]
#     versus.integrations.jira()                     → UnifiedJiraClient
#         .create_issue(summary, description)        → dict
#         .search_issues(jql)                       → list

```

```

#
# Confluence:
#     versus.integrations.confluence()           → UnifiedConfluenceClient
#     .search_pages(query)                       → list
#     .create_page(space_key, title, body)        → dict
#
# ServiceNow:
#     versus.integrations.snow_list()             → [{id, name, url}]
#     versus.integrations.snow()                 → ServiceNowClient
#     .get_info(number)                          → dict
#     .create_incident(short_description, description) → dict
#
# AI Providers (Anthropic, OpenAI, Azure OpenAI, Bedrock):
#     versus.integrations.ai_list()              → [{id, name, provider_type}]
#     versus.integrations.ai()                   → UnifiedAIClient
#     .complete(prompt, system='')               → str
#     .summarise(text)                           → str
#
# Cloud Accounts (AWS, Azure, GCP):
#     versus.integrations.cloud_list()            → [{id, name, provider}]
#     versus.integrations.cloud()                → UnifiedCloudClient
#     .list_instances(state='running')           → list
#     .list_load_balancers()                     → list
#
# Certificates (Versus-managed TLS):
#     versus.integrations.certs()                 → CertAuthoritySDK
#     .list_expiring(days=30)                    → list
#     .parse_pem(pem)                            → dict
#
# Cert Platforms (Venafi, Keyfactor, AppViewX):
#     versus.integrations.cert_platform_list()    → [{id, name, platform_type}]
#     versus.integrations.cert_platform()         → UnifiedCertPlatformClient
#     .list_expiring(days=30)                    → list
#     .renew_certificate(cert_id)                 → dict
#
versus.log(*args)
# Write a message to the workflow step log.
#
versus.ping(device_id, target, count=4)
# Run a ping from a managed device to a target IP or hostname.
#
# Detects vendor and runs appropriate CLI ping syntax:
#     F5 BIG-IP → run /util ping -c N target
#     Cisco/NXOS → ping target repeat N
#     Arista EOS → ping target repeat N
#     Juniper → ping target count N rapid
#     Palo Alto → ping host target
#     Generic → ping -c N target
#
# Args:
#     device_id – Device.id
#     target – IP address or hostname to ping
#     count – number of echo requests (default 4)
#
# Returns raw CLI output. Parse for '!!!!', 'loss', 'unreachable'.
#
versus.probe(device_id, kind, **args)
# Run one native vendor troubleshooting probe against a device using its
# own API/CLI driver – the same drivers Diagnostics AutoPilot uses.
#
# kind – one of versus.available_probes(device_id); common kinds:
#     'health' reachability + software version
#     'ha' HA / cluster state and peers
#     'stats' throughput / connection counters
#     'vs_status' virtual server / listener status (pass vs_name=)
#     'pool_status' pool / service-group member health (pass pool_name=)
#     'objects' config objects (rules, servers, nodes, policies...)
#     'certs' installed certificates
#     'interfaces' interface state
#     'events'/'sessions'/'cpu'/'bgp' where the vendor supports it
#

```

```

# Returns a formatted string, or None when the vendor has no native driver
# for this probe (fall back to versus.run_ssh(device_id, '<cli>')).

versus.query_snow_changes(start_dt, end_dt, category: str | None = None) -> list
# Return change requests overlapping the given datetime window.
#
# Args:
#     start_dt - datetime (aware or naive UTC)
#     end_dt   - datetime (aware or naive UTC)
#     category - optional SNOW category filter, e.g. 'Network'
#
# Returns list of dicts: {number, short_description, description,
#                        state, start_date, end_date, category, assigned_to}

versus.run_ssh(device_id, command, timeout=30)
# Run an SSH command on any managed device using its stored credential.
#
# Works with Cisco, Arista, FortiGate, Juniper, Palo Alto, and any other
# SSH-capable device. Returns the combined stdout output as a string.
#
# Args:
#     device_id - Device.id
#     command   - CLI command string, e.g. 'show version'
#     timeout   - seconds to wait (default 30)
#
# Returns raw output string.
# Raises RuntimeError if no credential is configured.

versus.run_tmsh(device_id, command)
# Run a TMSH command on an F5 BIG-IP device.
#
# Args:
#     device_id - Device.id of the target BIG-IP
#     command   - TMSH command string, e.g. 'show /ltm virtual'
#
# Returns the raw command output as a string.
# Raises RuntimeError if the device has no credential configured.

versus.set_output(key, value)
# Set a variable that downstream steps can read with get_input().

versus.traceroute(device_id, target)
# Run a traceroute from a managed device to a target.
#
# Args:
#     device_id - Device.id
#     target    - IP address or hostname
#
# Returns raw traceroute output showing hops and latency per hop.

versus.update_vuln_result(device_id, cve_id, status, evidence='')
# Update a DeviceVulnerabilityResult record for this device + CVE.
#
# Args:
#     device_id - Device.id
#     cve_id    - e.g. 'CVE-2024-12345'
#     status    - 'confirmed' | 'patched' | 'not_affected' | 'potentially_affected'
#     evidence  - optional text explaining the determination

versus.vendor_family(device_id)
# Native-driver family detected for a device - e.g. 'netscaler', 'panos',
# 'fortigate', 'checkpoint', 'aci', 'asa', 'avi', 'nsxt', 'cloudflare' -
# or None when no native driver fits (use versus.run_ssh() instead).

versus.virtual_servers(device_id=None, name=None, status=None, limit=200)
# Return a list of virtual servers.
#
# Returns list of dicts:
#     {id, name, destination, port, protocol, status, enabled,
#      device_id, device_hostname}

versus.wideips(device_id=None, name=None, enabled=None, limit=200)
# Return a list of GTM WideIPs.
#
# Returns list of dicts:

```

```
# {id, name, partition, record_type, pool_lb_mode,
# enabled, device_id, device_hostname, pool_count}
```

#### # — Integrations — versus.integrations.<method> —————

```
# One accessor per provider returns a client; each client's own methods are
# listed below. Example: versus.integrations.slack().message("deploy done").
```

##### versus.integrations.ai(name: str | None = None)

```
# Return a UnifiedAIClient for the named (or default) AI provider.
```

```
#
```

```
# If name is omitted, returns the provider marked is_default=True.
```

```
# Raises ValueError if a named provider is not found.
```

```
# Raises RuntimeError if no providers are configured at all.
```

```
#
```

```
# UnifiedAIClient methods:
```

```
# .complete(prompt, system='', max_tokens=2048) → str
```

```
# Single-turn prompt/response.
```

```
# .chat(messages, system='', max_tokens=2048) → str
```

```
# Multi-turn; messages = [{role, content}, ...]
```

```
# .extract(text, instruction, max_tokens=1024) → str
```

```
# Extract / classify information from a block of text.
```

```
# .summarise(text, max_tokens=512) → str
```

```
# Concise 2-4 sentence summary.
```

```
# .classify(text, labels, max_tokens=64) → str
```

```
# Classify text into one of the given label strings.
```

```
# .test_connection() → (bool, str)
```

```
# .name / .provider_type / .model / .is_default
```

##### versus.integrations.ai\_list() -> list[dict]

```
# Return all enabled AI providers.
```

```
# Each dict: {id, name, provider_type, model, is_default, status}
```

##### versus.integrations.appviewx(name: str | None = None)

```
# Return a UnifiedCertPlatformClient for AppViewX CERT+.
```

```
#
```

```
# Equivalent to integrations.cert_platform(platform_type='appviewx').
```

```
#
```

```
# See integrations.cert_platform() for full method reference.
```

```
#
```

```
# Example – trigger a cert push workflow after F5 upgrade:
```

```
# avx = integrations.appviewx()
```

```
# avx.trigger_workflow('Post-Upgrade-Cert-Refresh', params={
```

```
#     'device': device.hostname,
```

```
#     'vip': '/Common/vs_443',
```

```
# })
```

##### versus.integrations.awx(name: str | None = None)

```
# Return a UnifiedAWXClient for the named (or first enabled) AWX/Tower source.
```

```
#
```

```
# Raises ValueError if a named source is not found.
```

```
# Raises RuntimeError if no sources are configured.
```

```
#
```

```
# UnifiedAWXClient methods:
```

```
#
```

```
# Connection:
```

```
# .test_connection() → (bool, str)
```

```
#
```

```
# Job Templates:
```

```
# .list_job_templates(name=None) → [{id, name, description, playbook, project, inventory, last_job_run,
last_job_failed}]
```

```
# .get_job_template(template_id) → template dict
```

```
# .find_job_template(name) → template dict (raises RuntimeError if not found)
```

```
#
```

```
# Launching:
```

```
# .launch_job_template(template_id, extra_vars=None, inventory=None,
```

```
# limit=None, credential_ids=None)
```

```
# → job dict {id, status, name, template_id, template_name, started, ...}
```

```
# .run_and_wait(template_id, extra_vars=None, inventory=None, limit=None,
```

```
# credential_ids=None, timeout=600, poll_interval=10)
```

```
# → final job dict (blocks until terminal state)
```

```
#
```

```
# Jobs:
```

```

# .get_job(job_id) → {id, status, name, template_id, started, finished, elapsed, failed}
# .list_jobs(template_id=None, status=None) → [job dicts]
# .cancel_job(job_id) → None
# .get_job_stdout(job_id) → str (plain text)
# .wait_for_completion(job_id, timeout=600, poll_interval=10) → final job dict
#
# Inventories:
# .list_inventories() → [{id, name, description, host_count, organization}]
#
# Projects:
# .list_projects() → [{id, name, scm_type, scm_url, scm_branch, status}]
# .sync_project(project_id) → dict
#
# Workflow Templates (AWX multi-playbook workflows):
# .list_workflow_templates() → [{id, name, description, last_job_run}]
# .launch_workflow_template(template_id, extra_vars=None, inventory=None) → dict
#
# Example – run a job template and check the result:
# awx = integrations.awx()
# result = awx.run_and_wait(
#     template_id=42,
#     extra_vars={'target_hosts': 'bigip-01.corp.com', 'action': 'deploy'},
#     timeout=600,
# )
# if result['status'] != 'successful':
#     output = awx.get_job_stdout(result['id'])
#     integrations.slack().send_message(
#         f"AWX deployment failed:\n```${output[-1000:]}```", channel='#netops'
#     )
#     raise RuntimeError(f"AWX job failed: {result['status']}")
#
# Example – find and launch by name:
# awx = integrations.awx('Production Tower')
# template = awx.find_job_template('F5 Config Deploy')
# job = awx.launch_job_template(template['id'], extra_vars={'env': 'prod'})
# versus.set_output('awx_job_id', job['id'])

```

**versus.integrations.awx\_list() -> list[dict]**

```

# Return all enabled Ansible Tower / AWX sources.
# Each dict: {id, name, url, status}

```

**versus.integrations.cert\_platform(name: str | None = None, platform\_type: str | None = None)**

```

# Return a UnifiedCertPlatformClient for the named (or first enabled) cert platform.
#
# Args:
#     name – source name (case-insensitive); if omitted uses first enabled
#     platform_type – optional filter: 'venafi_cloud' | 'venafi_tpp' | 'keyfactor' | 'appviewx'
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# UnifiedCertPlatformClient methods (normalised across all platforms):
#
# Discovery:
# .list_zones()
#     → zones / templates / CAs available for certificate issuance
# .list_workflows_available()
#     → AppViewX/Keyfactor workflow list
#
# Certificate management:
# .list_certificates(limit=50, query='')
#     → [{id, commonName, notAfter, issuer, status, ...}]
# .list_expiring(days=30, limit=100)
#     → certificates expiring within N days
# .get_certificate(cert_id)
#     → full cert dict
# .request_certificate(common_name, zone_or_policy, sans=[], validity_days=365, ...)
#     → enrollment/request response
# .renew_certificate(cert_id)
#     → renewal response

```

```

# .revoke_certificate(cert_id, reason='Superseded')
#     → revocation response
#
# Workflow triggering:
# .trigger_workflow(name_or_policy, params={})
#     → Venafi TPP: POST /vedsdk/certificates/checkpolicy
#     → Keyfactor: start orchestrator workflow by name
#     → AppViewX: execute named CERT+ workflow
#
# Example – request a cert from Venafi Cloud:
# venafi = integrations.venafi()
# req = venafi.request_certificate(
#     common_name = 'api.corp.com',
#     zone_or_policy = 'Default',
#     sans = ['api.corp.com', 'api-lb.corp.com'],
# )
# integrations.slack().send_message(
#     f"Cert requested for api.corp.com – request ID: {req.get('id')}")
#
# Example – renew all certs expiring in 14 days:
# kf = integrations.keyfactor()
# for cert in kf.list_expiring(days=14):
#     kf.renew_certificate(cert['Id'])
#     ctx.log(f"Renewed: {cert['IssuedCN']} (exp {cert['NotAfter']})")
#
# Example – trigger AppViewX workflow to push renewed cert to ADC:
# avx = integrations.appviewx()
# avx.trigger_workflow('Push-Cert-to-F5', params={
#     'device': 'bigip-01.corp.com',
#     'partition': 'Common',
#     'cert_cn': 'api.corp.com',
# })
#
versus.integrations.cert_platform_list() -> list[dict]
# Return all enabled cert platform sources.
# Each dict: {id, name, platform_type, status}
#
versus.integrations.certs()
# Return a CertAuthoritySDK instance for certificate operations.
#
# No configuration needed – uses the certificates already configured
# in Versus under TLS Certificates → Authorities.
#
# CertAuthoritySDK methods:
#
# CAs:
# .list_cas(public_only=None) → [{id, name, ca_type, is_public, is_default, ...}]
# .get_ca(name) → CA dict
#
# Managed certificates:
# .list_certs(status=None, ca_name=None) → [{id, common_name, status, ca, not_after, ...}]
# .list_expiring(days=30) → [{id, common_name, status, not_after, days_until_expiry, ...}]
# .list_expired() → [{id, common_name, status, not_after, ...}]
# .get_cert(common_name) → cert dict with pem_certificate + pem_chain
# .get_cert_by_id(id) → cert dict
# .get_deployments(cert_id) → [{device, profile_name, status, deployed_at}]
#
# PEM inspection (works on any PEM string, not just Versus-managed):
# .parse_pem(pem) → {subject_cn, sans, issuer_cn, not_before, not_after,
#     serial_number, fingerprint_sha256, key_type, key_bits,
#     is_ca, is_self_signed, days_until_expiry, is_expired,
#     is_expiring_soon}
# .check_expiry(pem) → {not_after, days_until_expiry, is_expired, is_expiring_soon}
#
# Issuance (Versus CA only for immediate issuance; ACME CAs create a pending record):
# .issue(common_name, sans=None, ca_name=None, key_type='rsa2048',
#     validity_days=None, organization='', country='', auto_renew=True)
#     → {id, common_name, status, ca, message}
#
# Example – alert on expiring certs:

```

```

#     certs = versus.integrations.certs()
#     slack = versus.integrations.slack()
#     expiring = certs.list_expiring(days=14)
#     if expiring:
#         names = ', '.join(c['common_name'] for c in expiring)
#         slack.alert(text=f'{len(expiring)} certs expiring in <14 days: {names}',
#                     severity='warning', title='Certificate Expiry Alert')
#
# Example – parse a cert retrieved from a device:
#     info = certs.parse_pem(device_cert_pem)
#     if info['days_until_expiry'] < 30:
#         jira.create_issue(f'Certificate renewal needed: {info["subject_cn"]}',
#                          f'Expires {info["not_after"]} ({info["days_until_expiry"]}d)')
versus.integrations.cloud(name: str | None = None, provider: str | None = None)
# Return a UnifiedCloudClient for the named (or first enabled) Cloud Account.
#
# Args:
#     name – source name (case-insensitive); if omitted, the first enabled
#           account is used
#     provider – optional filter when name is omitted: 'aws' | 'azure' | 'gcp'
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no accounts are configured.
#
# UnifiedCloudClient methods (work across AWS / Azure / GCP):
#
# Account identity:
#     .test_connection() → (bool, str)
#     .get_account_info() → {account, arn, ...} or provider equivalent
#
# Compute:
#     .list_instances(region=None, state='running', **kwargs)
#         → [{id, name, state/status, private_ips, public_ips, ...}]
#     .get_instance(instance_id, region=None, **kwargs) → instance dict
#     .start_instance(instance_id, ...) → dict
#     .stop_instance(instance_id, ...) → dict
#     .reboot_instance(instance_id, ...) → None
#
# Networking:
#     .list_networks(region=None, **kwargs) → [{name, ...}]
#     .list_subnets(region=None, **kwargs) → [{name, prefix/cidr, ...}]
#     .list_load_balancers(region=None, **kwargs) → [{name, dns_name, ...}]
#     .list_security_groups(region=None, **kwargs) → [security group / NSG / firewall dicts]
#     .get_security_group(sg_id, ...) → dict
#
# Object storage:
#     .list_buckets(**kwargs) → [{name, location, ...}]
#     .list_objects(bucket, prefix='') → [{name/key, size, ...}]
#     .get_object_text(bucket, key) → str
#
# Secrets / parameters:
#     .get_secret(*args, **kwargs) → str (SSM / Key Vault / Secret Manager)
#     .list_secrets(*args, **kwargs) → [{name, ...}]
#
# Metrics:
#     .get_metric(metric_name, ...) → [{timestamp, value}] – provider-specific args
#     .list_alarms(region=None, **kwargs) → [{name, state, ...}]
#
# DNS:
#     .list_dns_zones(**kwargs) → zone list
#     .list_dns_records(zone_id, **kwargs) → record list
#     .upsert_dns_record(zone_id, name, type, values, ttl=300) → None (AWS only)
#
# Provider-specific pass-through:
#     .aws → raw AWSClient (only when provider='aws')
#     .azure → raw AzureClient (only when provider='azure')
#     .gcp → raw GCPClient (only when provider='gcp')
#

```

```
# Example – check for stopped instances and alert:
#     aws = integrations.cloud('AWS Prod')
#     stopped = [i for i in aws.list_instances(state='stopped')]
#     if stopped:
#         names = ', '.join(i['name'] for i in stopped)
#         integrations.slack().send_message(f'Stopped EC2: {names}', channel='#cloud-ops')
#
# Example – retrieve a secret and use it in automation:
#     cloud = integrations.cloud('AWS Prod')
#     db_pass = cloud.get_secret('/prod/db/password')
#
# Example – cross-cloud health check across all accounts:
#     for acct in integrations.cloud_list():
#         c = integrations.cloud(acct['name'])
#         ok, msg = c.test_connection()
#         if not ok:
#             integrations.pager().critical(f'Cloud auth failure: {acct["name"]}', body=msg)
```

**versus.integrations.cloud\_list() -> list[dict]**

```
# Return all enabled Cloud Account sources.
# Each dict: {id, name, provider, account_id, status}
```

**versus.integrations.confluence(name: str | None = None)**

```
# Return a UnifiedConfluenceClient for the named (or first enabled) Confluence source.
#
# Supports Confluence Cloud and Confluence Server/Data Center.
# Cloud API v2 and Server API v1 are handled transparently.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# UnifiedConfluenceClient methods:
#
# Spaces:
#     .list_spaces(limit=50)      → [{id, key, name, type}]
#     .get_space(space_key=None) → space dict
#
# Pages:
#     .get_page(page_id)          → {id, title, body_storage, version, space_key, url}
#     .search_pages(query, space_key=None, limit=25) → [{id, title, space_key, url}]
#     .list_pages_in_space(space_key=None, limit=50) → [{id, title, space_key, url}]
#     .create_page(space_key, title, body, parent_id=None) → {id, title, space_key, url}
#     .update_page(page_id, title, body)      → {id, title, ...}
#     .append_to_page(page_id, html_block)    → {id, title, ...}
#     .delete_page(page_id)
#
# Labels / Comments:
#     .add_label(page_id, label)
#     .add_comment(page_id, html_body)
#
# .test_connection() → (bool, str)
# .name / .deployment / .default_space
```

**versus.integrations.confluence\_list() -> list[dict]**

```
# Return all enabled Confluence sources.
# Each dict: {id, name, deployment, url, default_space, status}
```

**versus.integrations.crowdstrike(name: str | None = None)**

```
# Return a UnifiedCrowdStrikeClient for the named (or first enabled) CrowdStrike source.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# Authentication: OAuth2 client-credentials. Token auto-refreshed every 30 minutes.
#
# UnifiedCrowdStrikeClient methods:
#
# Connection:
#     .test_connection()          → (bool, str)
#
# Hosts:
```

```

# .search_hosts(filter_fql=None, limit=100) → list[str] (AIDs)
# .get_host_details(aids: list[str]) → list[dict]
# .find_host(hostname: str) → dict | None
# .list_hosts(filter_fql=None, limit=100) → list[dict]
# Host dict: {aid, hostname, platform, os_version, agent_version, status,
#             containment_status, external_ip, local_ip, mac_address,
#             first_seen, last_seen, groups, tags}
#
# Containment (network isolation):
# .contain_hosts(aids: list[str]) → dict
# .lift_containment(aids: list[str]) → dict
#
# Detections:
# .search_detections(filter_fql=None, limit=50) → list[str] (detection IDs)
# .get_detection_details(detection_ids) → list[dict]
# .list_detections(filter_fql=None, limit=50) → list[dict]
# .update_detection(detection_id, status, assigned_to_uuid=None, comment=None)
#     status values: new | in_progress | true_positive | false_positive | ignored | closed | reopened
#
# Alerts:
# .search_alerts(filter_fql=None, limit=50) → list[str]
# .list_alerts(filter_fql=None, limit=50) → list[dict]
# .update_alerts(alert_ids, action_params) → dict
#
# Incidents:
# .search_incidents(filter_fql=None, limit=50) → list[str]
# .list_incidents(filter_fql=None, limit=50) → list[dict]
#
# Spotlight (Vulnerabilities):
# .search_vulnerabilities(filter_fql=None, limit=100) → list[str]
# .get_vulnerability_details(vuln_ids) → list[dict]
# .get_host_vulnerabilities(aid, limit=100) → list[dict]
# Vuln dict: {id, aid, hostname, cve_id, severity, severity_name, description,
#             status, remediation, created, updated}
#
# Real Time Response (RTR):
# .run_rtr_script(aid, script, timeout=120) → str (stdout)
#     High-level: opens session, runs script, returns output, closes session.
# .init_rtr_session(aid) → session_id
# .run_rtr_command(session_id, command, base_command='runscript', timeout=120) → dict
# .get_rtr_result(cloud_request_id, sequence_id=0) → dict
# .delete_rtr_session(session_id) → None
#
# Policies & Groups:
# .list_prevention_policies() → list[dict]
# .list_host_groups() → list[dict]
# .get_host_group_members(group_id) → list[str] (AIDs)
#
# FQL (Falcon Query Language) filter examples:
# "hostname:'BIGIP-01'"
# "status:'contained'"
# "platform_name:'Windows'"
# "tags:'CriticalInfra'"
# "status:'new'+max_severity_displayname:'Critical'"
# "aid:'abc123'+status:'open'" ← Spotlight vuln filter
#
# Example – contain a host on detection of a critical finding:
# cs = integrations.crowdstrike()
# host = cs.find_host('bigip-01.corp.com')
# if host and host['containment_status'] == 'normal':
#     cs.contain_hosts([host['aid']])
#     integrations.pager().critical(
#         f"CrowdStrike: containing {host['hostname']}",
#         body=f"AID: {host['aid']}"
#     )
#     versus.set_output('contained_aid', host['aid'])
#
# Example – pull Spotlight CVEs for an asset:
# cs = integrations.crowdstrike()

```

```

#     host = cs.find_host(versus.get_input('target_hostname'))
#     vulns = cs.get_host_vulnerabilities(host['aid'])
#     critical = [v for v in vulns if v['severity_name'] == 'CRITICAL']
#     versus.log(f"{len(critical)} critical CVEs on {host['hostname']}")

versus.integrations.crowdstrike_list() -> list[dict]
#     Return all enabled CrowdStrike sources.
#     Each dict: {id, name, base_url, status}

versus.integrations.ipam(name: str | None = None)
#     Return a UnifiedIPAMClient for the named (or first enabled) IPAM source.
#
#     Vendors supported: Infoblox, NetBox, phpIPAM, BlueCat, EfficientIP.
#     The interface is identical regardless of vendor.
#
#     Raises ValueError if a named source is not found.
#     Raises RuntimeError if no sources are configured at all.
#
#     UnifiedIPAMClient methods:
#
#     Networks:
#     .get_networks(max_results=500)
#         → [{network, name, comment, utilization, vrf}]
#     .find_network(cidr_or_partial)
#         → filtered subset of get_networks()
#
#     IPs in a subnet:
#     .get_network_ips(network_cidr, max_results=500)
#         → [{ip, hostname, comment, status, ref}]
#     .count_used_ips(network_cidr) → int
#
#     Next-available:
#     .get_next_available_ip(network_cidr) → str    (does NOT allocate)
#     .get_next_available_ips(network_cidr, count=5) → [str, ...]
#
#     Allocation / release:
#     .allocate_ip(network_cidr, fqdn, comment='') → {ip, ref, fqdn}
#         Keep `ref` – pass it to release_ip() to de-allocate.
#     .release_ip(ref) → None
#
#     DNS / host records:
#     .search_host(fqdn)                → [{ip, hostname, comment, status, ref}]
#     .add_a_record(fqdn, ip, comment='') → ref str
#     .delete_record(ref)                → None
#
#     Utilization:
#     .subnet_utilization(network_cidr)
#         → {network, total_hosts, used, available, utilization_pct}
#
#     .test_connection() → (bool, str)
#     .name / .vendor    – source name and type string

versus.integrations.ipam_list() -> list[dict]
#     Return all enabled IPAM sources.
#     Each dict: {id, name, source_type, status}

versus.integrations.jira(name: str | None = None)
#     Return a UnifiedJiraClient for the named (or first enabled) Jira source.
#
#     Supports Jira Cloud (REST v3 / ADF bodies) and Jira Server/DC (REST v2 / Markdown).
#
#     Raises ValueError if a named source is not found.
#     Raises RuntimeError if no sources are configured.
#
#     UnifiedJiraClient methods:
#
#     Projects:
#     .list_projects(limit=50)           → [{id, key, name, type}]
#     .get_project(project_key=None)     → project dict
#     .get_issue_types(project_key=None) → [str]

```

```

# Issues:
# .create_issue(summary, description='', project_key=None, issue_type=None,
#               priority='', labels=None, assignee_account_id='',
#               components=None, custom_fields=None) → {id, key, url}
# .get_issue(issue_key) → {id, key, summary, status, priority, assignee, labels, description, url}
# .update_issue(issue_key, summary=None, description=None, priority=None, labels=None)
# .add_comment(issue_key, body) → {id, body, created}
# .transition_issue(issue_key, status_name)
# .close_issue(issue_key)
# .reopen_issue(issue_key)
# .search_issues(jql, max_results=25) → [{id, key, summary, status, priority, url}]
# .list_open_issues(project_key=None, max_results=25) → [{...}]
# .attach_file(issue_key, filename, content, content_type='text/plain')
# .delete_issue(issue_key)
#
# .test_connection() → (bool, str)
# .name / .deployment / .default_project / .default_issue_type

versus.integrations.jira_list() -> list[dict]
# Return all enabled Jira sources.
# Each dict: {id, name, deployment, url, default_project, status}

versus.integrations.keyfactor(name: str | None = None)
# Return a UnifiedCertPlatformClient for Keyfactor Command.
#
# Equivalent to integrations.cert_platform(platform_type='keyfactor').
#
# See integrations.cert_platform() for full method reference.

versus.integrations.pager(name: str | None = None)
# Return a UnifiedPagerClient for the named (or first enabled) pager source.
#
# Vendors supported: PagerDuty · Opsgenie · Splunk On-Call (VictorOps) · xMatters · OnPage
# The interface is identical regardless of vendor.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured at all.
#
# UnifiedPagerClient methods:
#
# Triggering:
# .trigger(title, body='', severity=None, source='versus',
#          component='', group='', dedup_key='')
# → dedup_key str (keep this – pass to resolve/acknowledge)
# .critical(title, body='', ...) → dedup_key (severity=critical shorthand)
# .warning(title, body='', ...) → dedup_key (severity=warning shorthand)
# .info(title, body='', ...) → dedup_key (severity=info shorthand)
# .trigger_and_resolve(title, ...) → dedup_key (open + immediately close)
#
# Resolving / acknowledging:
# .resolve(dedup_key) → None (close the incident)
# .acknowledge(dedup_key) → None (ack/silence without closing)
#
# Reading (requires REST api_key for most vendors):
# .get_incidents(status='triggered', max_results=25)
# → [{id, title, status, severity, created_at, url}]
#
# .test_connection() → (bool, str)
# .name / .vendor / .default_severity

versus.integrations.pager_list() -> list[dict]
# Return all enabled pager sources.
# Each dict: {id, name, vendor, default_severity, status}

versus.integrations.rundeck(name: str | None = None)
# Return a UnifiedRundeckClient for the named (or first enabled) Rundeck source.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# UnifiedRundeckClient methods:
#

```

```

# System:
#     .test_connection()          → (bool, str)
#     .system_info()             → {version, build, ...}
#
# Projects:
#     .list_projects()           → [{name, description, url}]
#     .get_project(project=None) → {name, description, url}
#
# Jobs:
#     .list_jobs(project=None, group_path=None, job_filter=None)
#         → [{id, name, group, project, description, href}]
#     .get_job(job_id)           → job dict
#     .find_job(job_name, project=None, group_path=None) → job dict
#     .export_job_yaml(job_id)   → YAML string
#
# Triggering (non-blocking – returns execution immediately):
#     .run_job(job_id, options=None, node_filter=None, as_user=None)
#         → {id, status, job_id, job_name, started_at, href}
#     .run_job_by_name(job_name, project=None, options=None, node_filter=None, ...)
#         → execution dict
#     .run_and_wait(job_id=None, job_name=None, project=None, options=None,
#         node_filter=None, timeout=300, poll_interval=3)
#         → final execution dict (blocks until done)
#
# Tracking:
#     .get_execution(execution_id) → {id, status, started_at, ended_at, duration, href}
#     .list_executions(job_id, status=None, max_results=20) → [execution dicts]
#     .list_running_executions(project=None) → [execution dicts]
#     .abort_execution(execution_id) → dict
#     .wait_for_completion(execution_id, poll_interval=3, timeout=300) → final execution dict
#
# Output:
#     .get_output(execution_id, offset=0, max_lines=500)
#         → {entries: [{time, node, step, log, level}], completed, total_size}
#     .get_full_output(execution_id) → [{time, node, step, log, level}]
#     .get_output_text(execution_id) → plain text string
#
# Nodes:
#     .list_nodes(project=None, node_filter=None)
#         → [{name, hostname, os_name, os_version, tags, description}]
#     .get_node(node_name, project=None) → node dict
#
# Key Storage:
#     .list_keys(path='/') → [{name, path, type, url}]
#
# SCM:
#     .get_scm_status(project=None, integration='export') → dict
#     .perform_scm_action(action, project=None, integration='export',
#         job_ids=None, commit_message='') → dict
#
# Example – run a job and check the result:
#     rd = integrations.rundeck()
#     result = rd.run_and_wait('Deploy ACL', options={'vlan': '100', 'device': 'bigip-01'})
#     if result['status'] != 'succeeded':
#         log = rd.get_output_text(result['id'])
#         integrations.pager().critical('Rundeck deploy failed', body=log[:1000])
#         raise RuntimeError(f"Job failed: {result['status']}")
#
# Example – trigger async job, notify on start:
#     exec_ = rd.run_job_by_name('Backup Configs')
#     integrations.slack().send_message(
#         f"Backup started (execution #{exec_['id']})", channel='#ops'
#     )

```

**versus.integrations.rundeck\_list() -> list[dict]**

# Return all enabled Rundeck sources.

# Each dict: {id, name, url, default\_project, api\_version, status}

**versus.integrations.scm(name: str | None = None)**

# Return a UnifiedSCMClient for the named (or first enabled) SCM source.

```

# Raises ValueError if no matching source is found.
# Raises RuntimeError if no sources are configured at all.
#
# UnifiedSCMClient methods:
#
# Repos:
#     .get_repo(repo)           → {name, full_name, default_branch, description, private, stars, url}
#     .list_repos(owner='')     → [{name, full_name, default_branch, description, private}]
#     .get_default_branch(repo) → str
#
# Branches:
#     .list_branches(repo)      → [{name, sha, protected}]
#     .branch_exists(repo, branch) → bool
#
# Commits:
#     .get_commits(repo, branch='', since_hours=48, per_page=50)
#         → [{sha, author, message, committed_at, url, files_changed}]
#     .recent_commits(repo, hours=24) → same schema (shorthand)
#     .get_commit(repo, sha)
#         → {sha, author, message, committed_at, url, files_changed, additions, deletions}
#     .get_diff(repo, base, head)
#         → {ahead_by, behind_by, files: [{filename, status, additions, deletions}]}
#     .get_last_commit_for_file(repo, path, branch='') → commit dict | None
#
# Files:
#     .get_file(repo, path, ref='') → str (decoded content)
#     .list_files(repo, path='', ref='') → [{name, path, type, size}]
#     .has_file(repo, path, ref='') → bool
#
# Issues:
#     .create_issue(repo, title, body='', labels=None) → {number, url, state}
#     .list_issues(repo, state='open', per_page=50) → [{number, title, state, url, author, created_at}]
#
# Pull / Merge Requests:
#     .create_pr(repo, title, head, base, body='') → {number, url, state}
#     .list_prs(repo, state='open', per_page=50) → [{number, title, state, url, head, base, author}]
#     .get_pr(repo, number) → {number, title, state, url, head, base, mergeable, merged, body}
#     .merge_pr(repo, number, merge_method='merge') → None
#     .add_pr_comment(repo, number, body) → {id, url}
#
# Tags:
#     .list_tags(repo) → [{name, sha}]
#     .create_tag(repo, tag_name, sha, message='') → None
#
# GitHub Actions:
#     .trigger_workflow(repo, workflow_id, ref='main', inputs=None) → None
#
#     .test_connection() → (bool, str)
#     .name / .scm_type

```

#### **versus.integrations.scm\_list() -> list[dict]**

```

# Return all enabled SCM sources.
# Each dict: {id, name, scm_type, status}

```

#### **versus.integrations.siem(name: str)**

```

# Return a UnifiedSIEMClient for the named SIEMSource.
# The interface is identical regardless of vendor (Splunk, Sentinel,
# Elastic, QRadar, Wazuh, Sumo Logic, Datadog).
#
# Raises ValueError if no enabled source with that name is found.
#
# UnifiedSIEMClient methods:
#     .get_alerts(hours=24, max_results=200, severity=None)
#         → [{title, severity, category, src_ip, dst_ip, hostname,
#             event_time, raw_data}]
#     .get_critical_alerts(hours=24, max_results=100) → same schema
#     .get_high_alerts(hours=24, max_results=100) → same schema
#     .search(query, hours=24, max_results=200)
#         → normalized events; query language is vendor-native
#         (SPL for Splunk, KQL for Sentinel, AQL for QRadar,

```

```

#         q-string for Wazuh, substring match for others)
# .count_by_severity(hours=24)
#     → {'critical': N, 'high': N, 'medium': N, 'low': N, 'info': N}
# .get_alerts_for_device(ip, hours=24, max_results=100)
#     → alerts where src_ip or dst_ip matches the given IP
# .test_connection() → (bool, str)
# .source_type / .vendor - e.g. 'splunk', 'sentinel', 'qradar'
# .name - configured source name

versus.integrations.siem_list() -> list[dict]
# Return all enabled SIEM sources.
# Each dict: {id, name, source_type, status}

versus.integrations.slack(name: str | None = None)
# Return a UnifiedSlackClient for the named (or first enabled) Slack source.
#
# Supports both bot token (Web API) and Incoming Webhook modes.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# UnifiedSlackClient methods:
#
# Messaging:
# .message(text, channel=None, thread_ts='') → {ts, channel}
# .alert(channel=None, text='', severity='info',
#     title='', details={}) → {ts, channel}
# .send_blocks(blocks, text='', channel=None) → {ts, channel}
# .webhook(text, blocks=None) → None
# .update_message(channel, ts, text='', blocks=None) → dict
# .delete_message(channel, ts)
#
# Reactions:
# .add_reaction(channel, ts, emoji)
# .remove_reaction(channel, ts, emoji)
#
# Channels:
# .list_channels() → [{id, name, is_private, topic, member_count}]
# .get_channel_id(name) → str
# .channel_history(ch) → [{ts, text, user, thread_ts}]
#
# Users:
# .list_users() → [{id, name, real_name, email, is_bot}]
# .get_user_id(email) → str
# .dm(email, text) → {ts, channel}
#
# Files:
# .upload_file(content, filename, channel=None, initial_comment='')
#
# Block Kit helpers:
# .header(text) .section(text) .divider()
# .fields(*texts) .context(*texts)
# .button(text, action_id) .actions(*buttons)
#
# .test_connection() → (bool, str)
# .name / .default_channel

versus.integrations.slack_list() -> list[dict]
# Return all enabled Slack sources.
# Each dict: {id, name, workspace_name, default_channel, status}

versus.integrations.snow(name: str | None = None)
# Return a live ServiceNowClient for the named instance (or the first
# configured instance if name is omitted).
#
# Raises ValueError if no matching instance is found.
# Raises RuntimeError if no instances are configured at all.
#
# ServiceNowClient methods:
# .is_approved(number) → (approved: bool, state_label: str, msg: str)
# .get_info(number) → {number, table, state, approval, short_description, sys_id, priority}

```

```
# .get_change_state(number) → {number, sys_id, state, state_label, approval, approved}
# .add_work_notes(number, notes) → None
# .close_change(number, close_notes, work_notes='') → None
# .create_change_request(short_description, description, category='Network',
#                         risk='3', impact='3', priority='4', ...) → {number, sys_id, state, ...}
# .create_incident(short_description, description, urgency='2', impact='2',
#                  category='Network', subcategory='Security', ...) → {number, sys_id, state, ...}
# .query_changes_in_window(start_dt, end_dt, category=None) → list[dict]
# .test_connection() → (bool, str)
```

**versus.integrations.snow\_list() -> list[dict]**

```
# Return all configured ServiceNow instances.
# Each dict: {id, name, url, auth_type}
```

**versus.integrations.venafi(name: str | None = None, variant: str | None = None)**

```
# Return a UnifiedCertPlatformClient for Venafi TLS Protect.
#
# Args:
#     name - source name; if omitted uses first enabled Venafi source
#     variant - 'cloud' or 'tpp'; if omitted uses first Venafi source found
#
# Equivalent to integrations.cert_platform(platform_type='venafi_cloud') or 'venafi_tpp'.
#
# See integrations.cert_platform() for full method reference.
```

**versus.integrations.vsphere(name: str | None = None)**

```
# Return a UnifiedVSphereClient for the named (or first enabled) vCenter.
#
# Raises ValueError if a named source is not found.
# Raises RuntimeError if no sources are configured.
#
# UnifiedVSphereClient methods:
#
# Inventory:
#     .get_vms(datacenter='') → [{name, moref, power_state, cpu_count, memory_mib}]
#     .get_powered_on_vms(datacenter='') → POWERED_ON VMs only
#     .get_powered_off_vms(datacenter='') → POWERED_OFF + SUSPENDED
#     .find_vm(name, datacenter='') → VMs matching name substring
#     .get_vm_detail(moref) → full hardware/config dict
#     .count_vms(datacenter='') → int
#     .count_by_power_state(datacenter='') → {'on': N, 'off': N, 'suspended': N}
#     .get_hosts(datacenter='')
#         → [{name, moref, power_state, connection_state, cpu_count, memory_mib}]
#     .get_connected_hosts(datacenter='') → CONNECTED hosts only
#     .count_hosts(datacenter='') → int
#     .inventory_summary(datacenter='')
#         → {vm_total, vm_on, vm_off, vm_suspended, host_total, host_connected}
#
# Power actions (accepts VM display name OR moref):
#     .power_on(name_or_moref) → None (hard power-on)
#     .shutdown(name_or_moref, force=False) → None (graceful via Tools / hard)
#     .reboot(name_or_moref, force=False) → None (graceful via Tools / hard)
#     .suspend(name_or_moref) → None (suspend to memory)
#
# Snapshots:
#     .list_snapshots(name_or_moref)
#         → [{id, name, description, created_at, is_current}]
#     .create_snapshot(name_or_moref, snapshot_name, description='',
#                     include_memory=False, quiesce=False) → snapshot_id str
#     .revert_snapshot(name_or_moref, snapshot_id) → None
#     .delete_snapshot(name_or_moref, snapshot_id,
#                      remove_children=False) → None
#
# .test_connection() → (bool, str)
# .name / .url
```

**versus.integrations.vsphere\_list() -> list[dict]**

```
# Return all configured vSphere/vCenter connections.
# Each dict: {id, name, url, status, enabled}
```

```

# — Native vendor troubleshooting – versus.probe(device_id, kind, **args) —
# Read-only; the same per-vendor drivers Diagnostics AutoPilot uses. Returns a
# string, or None when the device has no native driver (then use versus.run_ssh()).
versus.probe(device_id, kind, **args)
versus.available_probes(device_id)
versus.vendor_family(device_id)
# probe kinds: health, ha, stats, vs_status, pool_status, interfaces, routes, arp, objects, certs, events, sessions, cpu,
bgp, vs_stats, capture
# per-vendor support:
#   a10: certs, ha, health, objects, pool_status, stats, vs_status
#   aci: events, ha, health, objects, stats
#   alteon: certs, ha, health, objects, pool_status, stats, vs_status
#   arista: bgp, capture, cpu, ha, health, interfaces, objects
#   asa: ha, health, objects
#   avi: certs, ha, health, objects, pool_status, stats, vs_status
#   aws_alb: health, pool_status, vs_status
#   azure_alb: health, vs_status
#   bluecat: health, objects
#   checkpoint: arp, bgp, capture, cpu, events, gateways, ha, health, interfaces, objects, packages, routes, rules,
sessions, stats
#   cisco: arp, bgp, cpu, health, interfaces, objects, ping, routes
#   cloudflare: events, health, objects
#   efficientip: events, health, objects
#   f5os: ha, health, interfaces, objects, slots, stats
#   f5xc: events, health, pool_status, stats, vs_status
#   fortigate: arp, bgp, capture, cpu, ha, health, interfaces, objects, routes, sessions
#   infoblox: ha, health, objects
#   kemp: certs, ha, health, pool_status, stats, vs_status
#   netscaler: certs, ha, health, objects, pool_status, sessions, stats, vs_stats, vs_status
#   nexus: arp, bgp, capture, cpu, health, interfaces, objects, ping, routes
#   nsx_fw: ha, health, objects, stats
#   nsxt: ha, health, objects
#   panos: arp, bgp, cpu, events, ha, health, interfaces, objects, routes, sessions, stats
#   checkpoint = Check Point (gateway CLI + Security-Management Web API); aci = Cisco ACI/APIC REST.

# — Rules —
# No import statements (json/datetime/re/math are pre-injected); no file/network
# I/O except through versus.*; versus.log() to print; versus.get_input(k) /
# versus.set_output(k, v) to pass data between steps.

```