ALTAIR

Altair PBS Professional 2022.1

Budgets Guide

You are reading the Altair PBS Professional 2022.1

Budgets Guide (BG)

Updated 7/20/22

Contact Us

For the most recent information, go to the PBS Works website, www.pbsworks.com, select "My PBS", and log in with your site ID and password.

Altair

Altair Engineering, Inc., 1820 E. Big Beaver Road, Troy, MI 48083-2031 USA www.pbsworks.com

Sales

pbssales@altair.com 248.614.2400

Please send any questions or suggestions for improvements to agu@altair.com.

Technical Support

Need technical support? We are available from 8am to 5pm local times:

Location	Telephone	e-mail
Australia	+1 800 174 396	anz-pbssupport@india.altair.com
China	+86 (0)21 6117 1666	pbs@altair.com.cn
France	+33 (0)1 4133 0992	pbssupport@europe.altair.com
Germany	+49 (0)7031 6208 22	pbssupport@europe.altair.com
India	+91 80 66 29 4500 +1 800 208 9234 (Toll Free)	pbs-support@india.altair.com
Italy	+39 800 905595	pbssupport@europe.altair.com
Japan	+81 3 6225 5821	pbs@altairjp.co.jp
Korea	+82 70 4050 9200	support@altair.co.kr
Malaysia	+91 80 66 29 4500 +1 800 425 0234 (Toll Free)	pbs-support@india.altair.com
North America	+1 248 614 2425	pbssupport@altair.com
Russia	+49 7031 6208 22	pbssupport@europe.altair.com
Scandinavia	+46 (0)46 460 2828	pbssupport@europe.altair.com
Singapore	+91 80 66 29 4500 +1 800 425 0234 (Toll Free)	pbs-support@india.altair.com
South Africa	+27 21 831 1500	pbssupport@europe.altair.com
South America	+55 11 3884 0414	br_support@altair.com
UK	+44 (0)1926 468 600	pbssupport@europe.altair.com

Contents

About PBS Documentation

The PBS Professional guides and release notes apply to the *commercial* releases of PBS Professional.

Document Conventions

`Abbreviation`

The shortest acceptable abbreviation of a command or subcommand is underlined

Attribute

Attributes, parameters, objects, variable names, resources, types

`Command`

Commands such as `qmgr` and `scp`

Definition

Terms being defined

`File name`

File and path names

`Input`

Command-line instructions

Method

Method or member of a class

`Output`

Output, example code, or file contents

Syntax

Syntax, template, synopsis

`Utility`

Name of utility, such as a program

Value

Keywords, instances, states, values, labels

Notation

Optional Arguments

Optional arguments are enclosed in square brackets. For example, in the `qstat` man page, the `-E` option is shown this way:

qstat [-E]

To use this option, you would type:

```
qstat -E
```

Variable Arguments

Variable arguments (where you fill in the variable with the actual value) such as a job ID or vnode name are enclosed in angle brackets. Here's an example from the `pbsnodes` man page:

pbsnodes -v <vnode>

To use this command on a vnode named "my_vnode", you'd type:

```
pbsnodes -v my_vnode
```

Optional Variables

Optional variables are enclosed in angle brackets inside square brackets. In this example from the `qstat` man page, the job ID is optional:

qstat [<job ID>]

To query the job named "1234@my_server", you would type this:

```
qstat 1234@my_server
```

Literal Terms

Literal terms appear exactly as they should be used. For example, to get the version for a command, you type the command, then "--version". Here's the syntax:

qstat --version

And here's how you would use it:

```
qstat --version
```

Multiple Alternative Choices

When there are multiple options and you should choose one, the options are enclosed in curly braces. For example, if you can use either "-n" or "--name":

```
{-n | --name}
```

List of PBS Professional Documentation

The PBS Professional guides and release notes apply to the *commercial* releases of PBS Professional.

PBS Professional Release Notes

Supported platforms, what's new and/or unexpected in this release, deprecations and interface changes, open and closed bugs, late-breaking information. For administrators and job submitters.

PBS Professional Big Book

All your favorite PBS guides in one place: *Installation & Upgrade, Administrator's, Hooks, Reference, User's*, *Programmer's, Cloud, Budget,* and *Simulate* guides in a single book.

PBS Professional Installation & Upgrade Guide

How to install and upgrade PBS Professional. For the administrator.

PBS Professional Administrator's Guide

How to configure and manage PBS Professional. For the PBS administrator.

PBS Professional Hooks Guide

How to write and use hooks for PBS Professional. For the PBS administrator.

PBS Professional Reference Guide

Covers PBS reference material: the PBS commands, resource, attributes, configuration files, etc.

PBS Professional User's Guide

How to submit, monitor, track, delete, and manipulate jobs. For the job submitter.

PBS Professional Programmer's Guide

Discusses the PBS application programming interface (API). For integrators.

PBS Professional Manual Pages

PBS commands, resources, attributes, APIs.

PBS Professional Licensing Guide

How to configure licensing for PBS Professional. For the PBS administrator.

PBS Professional Cloud Guide

How to configure and use the PBS Professional Cloud feature in order to burst jobs to the cloud.

PBS Professional Budgets Guide

How to configure Budgets and use it to track and manage resource usage by PBS jobs.

PBS Professional Simulate Guide

How to configure and use the PBS Professional Simulate feature.

Where to Keep the Documentation

If you're not using the *Big Book*, make cross-references work by putting all of the PBS guides in the same directory.

Ordering Software and Licenses

To purchase software packages or additional software licenses, contact your Altair sales representative at pbssales@altair.com.

1 Introduction to Budgets

1.1 Using Budgets to Track, Reveal, and Manage Resource Use

Budgets allows you to track and manage credit and other resources for jobs at PBS complexes. You can define and track multiple currencies, and these can be any form of currency you need: dollars, CPU hours, GPU hours, etc. You can define how you want each currency calculated separately for on premise and cloud jobs at each PBS complex .

Budgets can provide a job submitter with an estimate for the cost of an on premise or cloud job. You can require that the owner of a job has sufficient credit to run a job, including checking credit before bursting cloud nodes.

Using Budgets, you can see how resources are being used at your site, and you can manage how those resources are used.

1.1.1 Two Modes: Postpaid and Prepaid

Budgets has two modes:

- In *postpaid mode*, Budgets tracks resource usage by users and projects, according to the criteria you define, but does not enforce limits. You use postpaid mode to understand how users and projects use resources, and how much they need. In postpaid mode, users do not need credit to run jobs. In postpaid mode, job submitters owe a positive number, similar to the way a credit card bill shows that a positive amount is owed.
- In *prepaid mode*, Budgets enforces usage limits; users need sufficient credit to run each job. You use prepaid mode to keep resource allocations within desired bounds. In prepaid mode, the amount of credit remaining in a job submitter's account is a positive number, and the amount debited for each job is a negative number.

The mode is global and applies to all of Budgets. You can switch back and forth between modes.

1.1.2 Using Postpaid Mode to Validate Jobs and Understand Resource Usage

Postpaid mode behaves as if your users and projects are charging everything to a credit card that never demands payment. They accumulate a credit balance. They can pay down the balance as they go, but that is not required for them to run jobs. The Budgets administrator can provide partial or full refunds if necessary.

This mode allows users and projects to run jobs without having been assigned a credit balance. However, you can use this balance as a basis for charging users, if your site charges users for resources.

Using postpaid mode, you can get detailed reporting about the amount of resources used by each user and project submitting jobs. In postpaid mode, Budgets provides reporting only for top-level time periods.

You can use postpaid mode to make sure that users and jobs are validated before jobs run.

In postpaid mode, Budgets adds the cost for a job to the job owner's account after the job runs.

1.1.2.1 Job Flow in Postpaid Mode

1. A job is submitted
2. Budgets checks the following:
    - Whether the submitter is associated with the cluster where the job is submitted
    - Whether any quotas have been exceeded
    - If a job is submitted as part of a project, whether the submitter is associated with the project

    If the submitter and job pass all of these tests, PBS allows the job to be enqueued
3. The job runs
4. After the job runs, Budgets adds the cost for the job to the job submitter or project credit balance

1.1.3 Using Prepaid Mode to Validate Jobs and Manage Credit and Costs

You represent each department in your organization as a group in Budgets. Each department receives investment which it can then spend on projects and individual users. Funding allocations apply to, are charged against, and are tracked in time periods. Every transaction is available for examination forever.

You can require that job owners have sufficient credit to run each job before allowing that job to be queued, or you can let jobs with insufficient credit be queued and remain there until the owner has sufficient credit to run them.

Before a job runs, Budgets checks that the job owner has enough currency to run the job, then transfers that amount from the job owner's account into escrow. After the job runs, Budgets reconciles charges for the job: it debits escrow for the amount consumed by the job and returns any excess to the job owner, or if the job consumed more than requested it debits more from the job owner's account. The Budgets administrator can provide partial or full refunds if necessary.

Budgets is flexible; a department can fund multiple projects and users working in multiple PBS complexes. A project or user can be funded by multiple departments and run jobs in multiple PBS complexes.

1.1.3.1 Managing Job Submission and Execution in Prepaid Mode

When you manage a PBS complex via Budgets in prepaid mode, each PBS job can run only when the job can be charged to a project or user account that has sufficient credit. The organization's CFO can fund a group representing a department or other organizational entity; this group can then deposit credit with project or user accounts.

A project spends credit when a user associated with the project runs a job and charges the job to the project account. A job can be charged to a project account only when it is run by an associated user on an associated complex. Any user associated with the project can charge a job to the project account.

An individual user spends credit when they run a job and charge it to their own account. This job must be run on a complex that is associated with the user.

When a group manager deposits group credit to a project account, that allocation is for a specific period and in a specific currency (service unit). Currency is usually defined as resource usage, such as CPU hours or GPU hours, but can be in dollars or other monetary units.

In addition to checking credit balances, Budgets checks whether a job owner has hit any quotas for externally-managed resources such as storage, using a mechanism called *dynamic service units*. See section 1.7.2.2, “Dynamic Service Units”, on page 19.

In prepaid mode, Budgets subtracts the estimated cost of each job from the job submitter or project credit balance before the job runs, then reconciles any difference after the job runs.

You can ensure that cloud jobs run only when the owner has sufficient credit. See section 3.2.7, "Requiring Sufficient Credit Before Bursting Cloud Nodes", on page 72.

1.1.3.2 Job Flow in Prepaid Mode

1. A job is submitted
2. Budgets checks the following:
   - Whether the submitter is associated with the cluster where the job is submitted
   - Whether any quotas have been exceeded
   - If a job is submitted as part of a project, whether the submitter is associated with the project
   - If AM_BALANCE_PRECHECK is set to *True* in the Budgets configuration file, whether the job submitter has sufficient credit to run the job
     - If there is sufficient credit, the job is enqueued
     - If there is not sufficient credit, the job is rejected by PBS
   - If AM_BALANCE_PRECHECK is set to *False* in the Budgets configuration file, whether the job submitter has sufficient credit to run the job
     - If there is sufficient credit, the job is enqueued
     - If there is not sufficient credit, the job is enqueued with a comment noting the insufficient credit
3. The scheduler selects the job to run
4. Budgets checks the following:
   - Whether the submitter is associated with the cluster where the job is submitted
   - Whether any quotas have been exceeded
   - If a job is submitted as part of a project, whether the submitter is associated with the project
   - Whether the job submitter has sufficient credit to run the job
     - If the submitter does not have enough credit, the job is returned to the queue
5. If the submitter has enough credit:
     - Budgets reserves the full amount of credit estimated to run the job
     - The job runs
6. After the job runs, Budgets reconciles the actual cost to run the job with the reserved credit, and returns any overage to the submitter

When AM_BALANCE_PRECHECK is set to *False* in the Budgets configuration file, Budgets keeps creditless jobs in the queue so that they will run as soon as the job submitter has sufficient credit; the job submitter does not need to resubmit or remove a hold from these jobs.

When AM_BALANCE_PRECHECK is set to *True* in the Budgets configuration file, PBS rejects any creditless jobs. These jobs must be resubmitted when the job submitter has sufficient credit.

1.1.4 Tracking Cloud Costs and On Premise Costs Separately

You can treat all jobs as if they will run on premise, using the same formulas for all jobs. However, you can instead use cloud-specific cost data in formulas designed specifically for cloud jobs, while using on premise formulas for on premise jobs. You can collect cloud instance cost data and give it to Budgets, and Budgets can use it when computing cloud-based billing costs. This way, you can track and charge for cloud costs separately from on premise costs. See section 3.2.6, “Separating On Premise and Cloud Costs”, on page 71.

You can use and manage cloud cost information separately from on premise cost information:

- You can keep Budgets informed about the cost per unit time for each cloud instance. You collect the cost information from the cloud provider, then use the `amgr update clouddata` command to give this information to Budgets. You may find it helpful to do this in a `cron` job or periodic hook. See section 4.2.3.10, “Updating Cloud Cost Data”, on page 98.
- You can see the cloud cost data that has been given to Budgets via the `amgr ls clouddata` command. See section 4.2.2.10, “Listing Cloud Data”, on page 90.
- You can create separate formulas for cloud-specific service units so that you can track cloud job costs. Budgets can use the data you gave it for each specific instance when calculating the cost for a job that will use that instance. You can also create formulas that combine on premise and cloud costs when you need to calculate total credit usage for a job owner. See section 3.2.3.3, “Defining Cloud and On Premise Service Units”, on page 65.
- You can specify that each cloud job can burst cloud nodes only when the job owner has sufficient credit, as calculated by Budgets. See section 3.2.6, “Separating On Premise and Cloud Costs”, on page 71.
- The administrator and the job owner can check whether that job owner has sufficient credit to run a job via the `amgr precheck` commands. See section 4.3.4, “Prechecking Service Unit Balance”, on page 124.

1.1.5 Recommendation: Start Using Budgets in Postpaid Mode

We recommend starting with postpaid mode, especially if you are already running jobs, so that you can get an idea of usage needs and patterns. In postpaid mode, you do not need to allocate credit to users and projects, and they can run jobs right away. If you need to enforce limits, you can switch to prepaid mode. In prepaid mode, you do need to allocate credit before jobs can be run.

If you want to begin using Budgets in prepaid mode but have not used such tools before, be aware that there are some steps to define the business processes you will use to manage credit, and some calibration to understand the amount of credit to provide to users and projects. It is important to get these steps right, because providing too little credit to users and projects could prevent the site workload from running.

We recommend that you begin by using Budgets in postpaid mode for a meaningful period of time (e.g. a quarter, especially if you manage credit per quarter) which will give you data you can use to guide your investments later in prepaid mode.

The easiest point at which to transition from postpaid to prepaid mode is on the boundary of a time period. For example if you are in Q1 and you have sufficient data to make your future investment decisions, you can make the transition between Q1 and Q2: before the end of Q1 you can pre-invest in Q2 users and projects, and then switch modes at the end of Q1.

1.2 Some Nuts and Bolts

1.2.1 The AMS Module

The AMS module provides the security framework for Budgets. Budgets uses AMS to retrieve tokens, verify them, and authenticate users. When you install Budgets, the Budgets server is registered as a client of AMS. Every Budgets request has an authorization token.

1.2.2 Number of Instances and Workers

A large site will probably find that a single instance of Budgets for all PBS complexes works best. With a single instance of Budgets, you can use a different charging model for each PBS complex.

Budgets is web-based for scalability. By default, the number of workers is two; you can spawn more if you need to.

1.2.3 Hooks and Formulas

Budgets is tightly integrated with the job management mechanisms of PBS Professional.

Budgets uses hooks to calculate billing at each PBS complex. Budgets uses two hooks, am_hook and am_hook_periodic, that are identical except for their triggering events, that are installed in the PBS complexes. These hooks calculate service unit consumption using the billing formulas defined in the formula file, which is used as the hook configuration file. You can define the formulas to be based on compute resources, time, day of week, time of day, priority of queue, etc. See section 3.2.3, “Define Billing Formulas”, on page 62.

1.2.4 Database

Budgets uses a Postgres database.

1.2.5 Budgets and PBS Cloud

Budgets and PBS Cloud are integrated so that Budgets can use cloud cost data to calculate costs for cloud jobs and track credit. Budgets can make sure that each cloud job owner has sufficient credit before allowing a cloud job to burst cloud nodes.

1.3 Budgets Terminology

Account

Budgets has user, project, and group accounts. See User, User Account, Project, Project Account, and Group, Group Account.

Accounting Policy

A policy associated with a project or user. Can be any of *begin_period*, *end_period*, or *proportionate*.

Active or inactive

An active element can perform all its normal functions; an inactive element cannot run jobs or do transactions.

Allocation

A specified amount of one or more service units designated for use by a project or user. An allocation exists for a fixed time period. As the service units are consumed, the allocation is depleted. Allocations can be made for compute resources, budget resources, time resources, or a combination of those.

AMS

Security module used by Budgets to retrieve tokens, verify them, and authenticate users.

Billing Formulas

An arithmetic formula defining how a service unit is charged at a PBS complex. This is typically a PBS resource and time, for example, the PBS resource ncpus multiplied by walltime defines the service unit for CPU hours. Other examples are GPU hours, memory hours, etc. The billing formulas for each PBS complex are defined in the Budgets hook configuration file for that complex. See section 3.2.3, “Define Billing Formulas”, on page 62.

Cluster

A data structure representing a PBS complex. Named for the PBS server. See section 1.7.5, “Clusters”, on page 23.

Element

Blanket term used to refer to a part of Budgets, such as an account, a transaction, a role, etc. Can be an entity or something associated with an entity, such as a limit on a service unit.

Entity

Data structure representing a part of Budgets: a period, cluster, service unit, user, project, or group.

Group, Group Account

Entity representing an organizational structure such as a department, business unit, etc. A group account has a credit balance, at least one associated investor who invests service units in the group account, and at least one associated manager who uses the group account to fund users and projects.

Instance

An installation of Budgets; described by a name and the hostname or IP address of the machine where Budgets is installed.

Mode

Budgets runs in either postpaid or prepaid mode. In postpaid mode, job submitters do not need credit to run jobs. In prepaid mode, job submitters need credit to run jobs. The mode is global and applies to all of Budgets.

PBS complex

An installation of PBS Professional consisting of daemons including a server daemon and at least one MoM daemon, and the hosts on which those daemons run. The name for a PBS complex is the name of its server daemon, which is found in PBS_SERVER parameter in the `/etc/pbs.conf` file.

Periods, Allocation Periods, Billing Periods

Time period. Budgets can use a hierarchical system of time periods.

Postpaid Mode

In postpaid mode, job submitters do not need credit in order to run jobs.

Prepaid Mode

In prepaid mode, job submitters must have sufficient credit in order to be able to run jobs.

Project, Project Account

Entity representing a project, for example a workflow, as well as its associated account, including its credit balance. A project has associated users and clusters, and users associated with the project can run jobs on those complexes and charge the project account.

Reconciling Jobs

Prepaid mode only. After a job runs, Budgets reconciles charges for the job: it debits escrow for the amount consumed by the job and returns any excess to the job owner, or if the job consumed more than requested it debits more from the job owner's account.

Roles

Roles define available actions and access to features in Budgets. Roles can be *admin*, *investor*, *manager*, *teller*, and *user*.

Service Unit

A standard service unit represents a currency, which can be dollars, CPU hours, GPU hours, etc; see section 1.7.2.1, “Standard Service Units”, on page 19.

A dynamic service unit lets you set a quota on an externally-managed resource such as storage; see section 1.7.2.2, “Dynamic Service Units”, on page 19.

Standard Service Units

Currency representing compute resources, budget resources, time resources, or a combination. Defined by the Budgets administrator; can be specific to the needs of the client. Examples: CPU hours, GPU hours, dollars.

Dynamic Service Units

Mechanism for placing a quota on an external compute resource such as storage. If a quota is exceeded, jobs that depend on that dynamic service unit do not start.

Transactions

An action that affects a credit balance, such as depositing or refunding service units to an account, or withdrawing service units from an account.

User, User Account

We use these terms to mean two different things:

- Budgets entity representing an individual user and their associated account, including its credit balance. This user is typically a job submitter with the *user* role, although an individual user can have any role. Each user account is funded by at least one group manager. See section 1.6.3, “User, User Account”, on page 16.
- Username, password, etc. for administrator, database user, teller, or job submitter. See section 2.5.1, “Create Required User Accounts”, on page 32.

Worker

Process that handles transactions for Budgets. This process is run by the web server.

1.4 Roles

Roles define available actions in and access to the features of Budgets. Except for the *teller* role, roles are hierarchical; each role includes the capabilities of the roles below it in the hierarchy. For example, if a user is defined as an *investor*, they can be added later to a group as a manager. A user can have only one role at a time, although that role may have the privileges of lower roles. Roles are case-sensitive and are lowercase. The *teller* role is a special-purpose role for processing transactions, and its capabilities are shared only with the *admin* role.

In hierarchical order, with the most powerful and inclusive role first:

admin

Budgets administrator. Configures Budgets; adds users, projects, groups, clusters, sets roles, defines periods, gives refunds, etc. Can modify any element in Budgets. Can transfer service units from any entity to any entity.

When you install Budgets, you specify a username for the Budgets administrator. This account automatically gets assigned the *admin* role. This username is typically *pbsadmin*.

Not automatically associated with specific projects or groups.

The *admin* role includes the capabilities of the *investor*, *manager*, *teller*, and *user* roles.

investor

A user who is associated with a group and is responsible for investing service units in the associated group account. Can also withdraw service units from the group account. An investor can be associated with one or more groups.

Created by Budgets administrator.

The *investor* role includes the capabilities of the *manager* and *user* roles.

manager

A user who is associated with a group and is responsible for depositing service units from the group account to associated user and project accounts. Can also withdraw service units from these user and project accounts, putting them back into the group account. Can be associated with multiple groups.

Created by Budgets administrator.

The *manager* role includes the capabilities of the *user* role.

teller

A user who performs all acquire and reconcile transactions on behalf of projects and users. When the Budgets hook performs these actions, it uses the *teller* role and username amteller.

The teller has full permissions for transactions and can read all projects, accounts, groups, etc.

Not associated with groups or projects.

The *teller* role is a special-purpose role for processing transactions, and its capabilities are shared only with the *admin* role.

Created by Budgets administrator.

The *teller* role includes the capabilities of the *user* role.

user

PBS job submitters who have an individual account and/or are associated with a project. A job submitter with an individual account can charge jobs to their own account. A job submitter who is associated with a project can charge jobs to the project.

Created by Budgets administrator.

Operations:

- When you create any user, you must assign a role to that user; see section 4.2.1.2, “Adding a User”, on page 80.
- To see all roles that have been created, list them; see section 4.2.2.9, “Listing Roles”, on page 89.
- To change a user's role, use `amgr update user`; see section 4.2.3.2, “Updating Users”, on page 93.

1.5 Investing and Consuming Service Units

Investing and consuming service units is required only in prepaid mode.

1.5.1 Investing in Groups (Cost Centers)

Groups in Budgets represent organizational entities such as departments or businesses. For example, an organization might have multiple departments including engineering, systems, and software. The CFO, who can be represented in Budgets as an investor, deposits service units to a department's group account. An investor can deposit to multiple groups, and a group can have multiple investors.

The amount each investor dispenses to a group is tracked. An investor can withdraw funds from a group, but cannot withdraw more than the amount they deposited to that group.

The group budget pool does not have any defined period. The amount in the budget pool is available for an infinite amount of time.

Operations:

- Investor invests in a group via `amgr deposit group`; see section 4.3.1.3, "Depositing Service Units to Group", on page 121.
- Investor withdraws from a group via `amgr withdraw group`; see section 4.3.3.3, "Withdrawing Service Units from Group", on page 124.
- Administrator transfers between groups via `amgr transfer group`; see section 4.3.8.5, "Transferring Service Units for Investors and Group", on page 134.

Figure 1-1 shows the basic path for investment/allocation of credit. Each arrow indicates a path for investment/allocation. A group investor puts credit in a group account, and a group manager allocates group credit to users and projects.

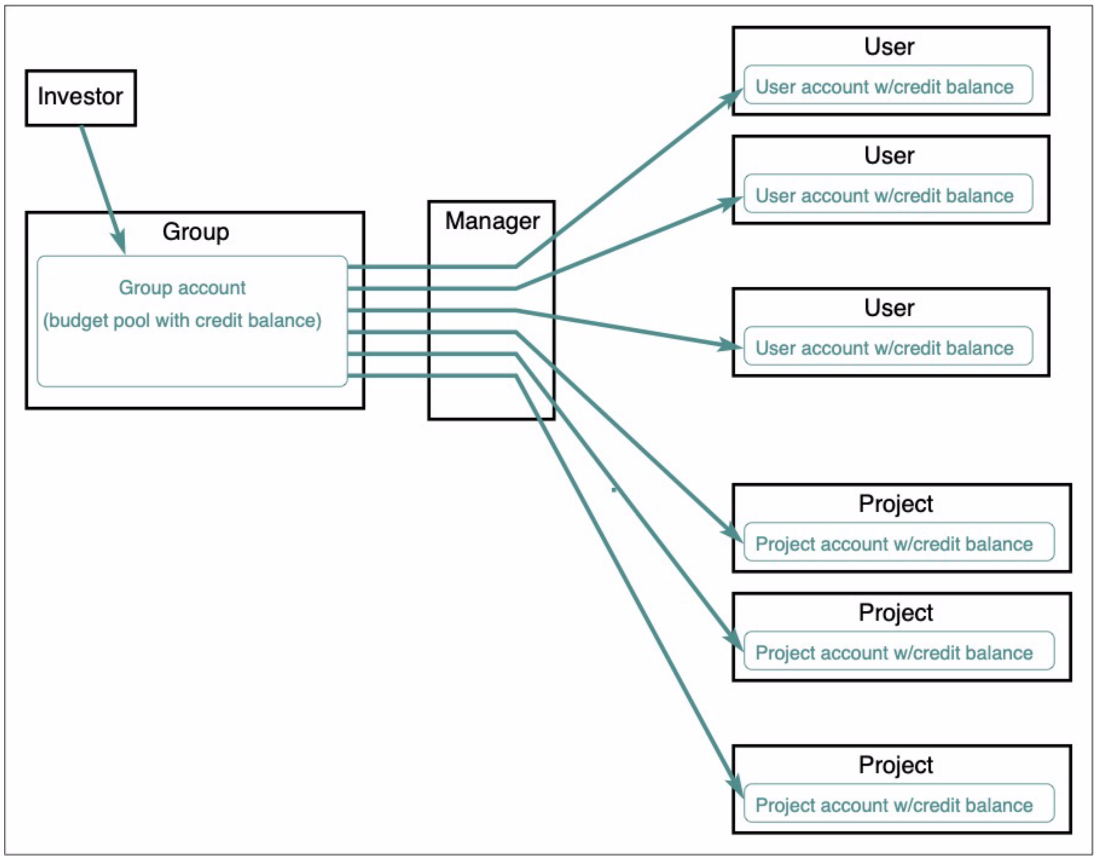
Investor
Group
Group account
(budget pool with credit balance)
Manager
User
User account w/credit balance
User
User account w/credit balance
User
User account w/credit balance
Project
Project account w/credit balance
Project
Project account w/credit balance
Project
Project account w/credit balance

Figure 1-1: Basic investment path

1.5.2 Investing in Users and Projects

Once the group has been been allocated service units, group managers can use that pool to fund projects and users. Each allocation to a user or project is for a specific period of time. For example, the engineering group managers can allocate funds to the design project for Quarter1 and to the testing project for Quarter2 and Quarter3.

A manager can use up to the entire pool on one user or project.

Each group manager can be linked to multiple groups, and each group can have multiple managers. Each group can fund multiple users and projects, and each user or project can have multiple groups funding it.

If necessary, a group manager can also withdraw funds from projects and users, but only up to the amount they deposited.

Figure 1-2 shows how credit can be allocated through multiple paths. Each arrow shows a potential investment path. An investor can be associated with multiple groups, a group can receive credit from multiple investors, a manager can be associated with multiple groups, a group can have multiple managers, and a user or project can be allocated credit from multiple groups and by multiple managers.

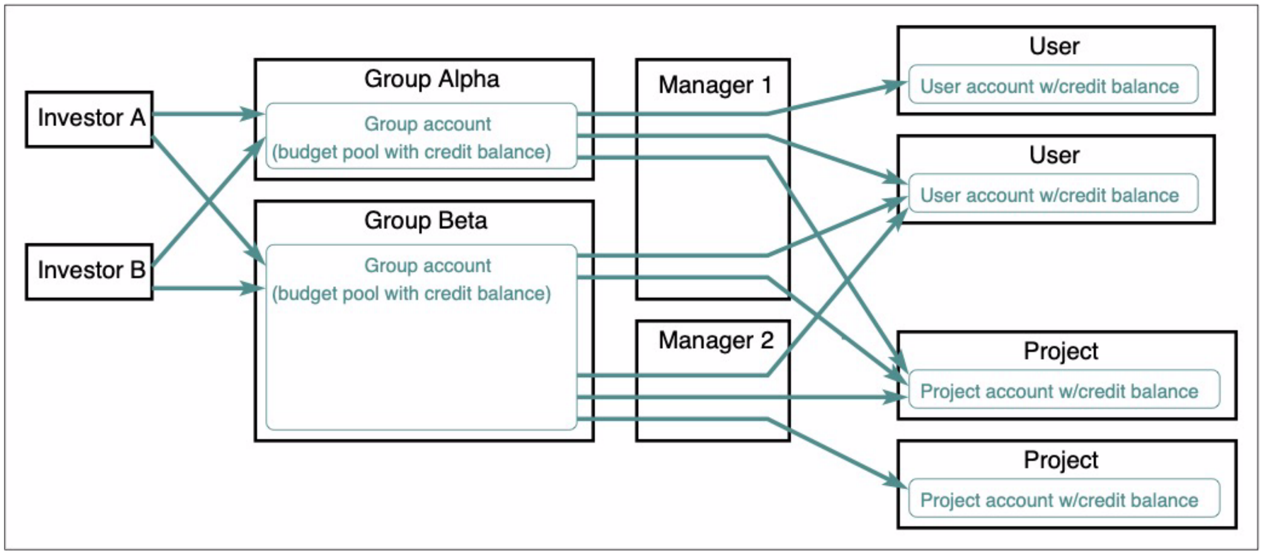
Investor A
Investor B
Group Alpha
Group account
(budget pool with credit balance)
Group Beta
Group account
(budget pool with credit balance)
Manager 1
Manager 2
User
User account w/credit balance
User
User account w/credit balance
Project
Project account w/credit balance
Project
Project account w/credit balance

Figure 1-2: Multiple investment paths

1.5.3 Charging Jobs to User or Project Account

The Budgets hook monitors each job submission, and when the job submitter specifies a project via `qsub -P <project name>`, Budgets charges the job to that project. If the job submitter does not specify a project, Budgets charges the job to the job submitter's own account.

1.5.4 Job and Credit Lifecycle

1.5.4.1 Job and Credit Lifecycle in Postpaid Mode

When a job runs, it can consume multiple kinds of service units, for example both CPU hours and GPU hours.

1. User submits a job, charging it either to the user's account or a project account
2. Budgets validates membership of the job owner (a user or project)
3. Job is queued at PBS server
4. Job runs at execution host(s)
5. When the job finishes, Budgets debits the amount of credit that was consumed by the job
6. After the job finishes, the administrator can optionally refund credit for the job

1.5.4.2 Job and Credit Lifecycle in Prepaid Mode

When a job runs, it can consume multiple kinds of service units, for example both CPU hours and GPU hours.

1. User submits a job, charging it either to the user's account or a project account
2. Budgets validates membership of the job owner (a user or project)
3. Optionally, Budgets validates credit of the job owner (a user or project)
4. Job is queued at PBS server
5. Before running the job:
    a. Budgets checks balance of job owner's account
    b. Budgets acquires service units from job owner and puts them in escrow
6. Job runs at execution host(s)
7. When the job finishes, Budgets reconciles the job: it debits the amount of credit that was consumed by the job and returns any excess to the user or project, or if the job consumed more than requested it debits more from the job owner's account
8. After the job finishes, the administrator can optionally refund credit for the job

1.5.5 Reconciling Jobs

For prepaid mode only. When a job finishes, Budgets reconciles the job by debiting the amount of credit used by the job, and returning the excess to the job owner's account, or if the job consumed more than requested it debits more from the job owner's account. However, in rare cases a job cannot be reconciled when the job fails. Budgets will try to reconcile a job up to 3 times; if it is not successful, it marks the job as not reconciled.

To see all unreconciled jobs that have had *count* or more attempts to reconcile:

amgr report transaction -i <job or transaction ID> -N <count>

For example, `amgr report transaction -N 2` displays all non-reconciled jobs with *count* >=2.

See section 4.2.5.5, “Getting Job and Transaction Reports”, on page 115.

You have to manually reconcile jobs with 3 failed attempts.

To reconcile a job that is part of a project:

amgr reconcile project -n <project name> -c <cluster> -f <formula file> -s <service unit name> <service unit amount> [-D <transaction date>] -u <job owner username> -i <transaction ID> -d <duration> [-C <comment>]

To reconcile a job that is not part of a project:

amgr reconcile user -n <username> -c <cluster> -f <formula file> -s <service unit name> <service unit amount> [-D <transaction date>] -u <job owner username> -i <transaction ID> -d <duration> [-C <comment>]

See section 4.3.6, “Reconciling Service Units”, on page 130.

1.5.6 Refunding Users and Projects

For postpaid and prepaid modes. The administrator can refund the user or project that funded a job for some or all of the service units charged for the job. You can choose to provide a refund for situations such as when a job fails or runs multiple times, for reasons which cannot be attributed to the job owner. You can provide refunds to active and inactive projects and users. You can provide multiple refunds for the same job, but the total refund cannot exceed the amount consumed by the job.

Budgets knows who paid for the job, so you do not have to specify where the refund goes. The refund amount is calculated by multiplying the specified percentage by the total consumed amount. Total consumed amount is the sum of all transaction amounts of all transactions for a job. Refunds are strictly per job.

You must be *admin* to provide a refund.

When a job is refunded, the comment in the report is always prefixed with "refund:".

See section 4.3.7, “Refunding Service Units”, on page 132.

1.5.7 Retrieving Abandoned Service Units

If service units go unclaimed and become unusable, the *admin* role can transfer them so that they become usable. You can transfer from any period, group, user, or project, to any other period, group, user, or project. You can transfer between investors in the same group or different groups. A transfer gets its own transaction ID.

Service units may become unusable when the following happen:

- An investor is unlinked from a group, leaving funds with the group
- A group is unlinked from a project, leaving funds with the project
- A period expires and leaves service units unused in a user or project account

See section 4.3.8, “Transferring Service Units”, on page 133.

You must be *admin* to run this command.

1.6 Accounts in Budgets

1.6.1 Group, Group Account

A group, also called a group account, is a type of account that represents an organizational structure such as a department, a business unit, etc. A group has the following:

- Name
- Credit balance
- One or more investors
- One or more managers
- Is active or inactive

Group managers provide the funding for users and projects to run jobs by depositing service units to those accounts. Group managers can also withdraw service units from associated job and project accounts and return the service units to the group account. All funding for users and projects comes from groups. The group credit balance is in standard service units; see Service Units.

Group managers can also set a quota on an externally-managed resource such as storage by setting a limit for the dynamic service unit representing that resource. A dynamic service unit limit is per user, project, and period, so you can set a different limit for each user and project for each period. Budgets checks usage by the job owner against this quota before running any job. See section 1.7.2.2, "Dynamic Service Units", on page 19.

Groups do not:

- Directly run jobs
- Have associated time periods
- Have associated dynamic service units

Operations:

- Administrator adds group via `amgr add group`; see section 4.2.1.4, "Adding a Group", on page 82
- Administrator updates group via `amgr update group`; see section 4.2.3.4, "Updating Groups", on page 95
- User lists group via `amgr ls group`; see section 4.2.2.4, "Listing Groups", on page 86
- Administrator removes group via `amgr rm group`; see section 4.2.4.4, "Removing a Group", on page 102. Note that if a group has any associated current or past transactions, removing it results only in making it inactive.
- Group manager deposits service units to users and projects via `amgr deposit`; see section 4.3.1, "Depositing Service Units", on page 119
- Group manager withdraws service units from users and projects via `amgr withdraw`; see section 4.3.3, "Withdrawing Service Units", on page 123
- Manager checks credit balance of group via `amgr checkbalance group`; see section 4.3.2.6, "Checking Service Unit Balance for Group", on page 122
- Group manager transfers service units to and from users and projects via `amgr transfer`; see section 4.3.8, "Transferring Service Units", on page 133
- Manager gets reports on groups via `amgr report group`; see section 4.2.5.4, "Getting Group Reports", on page 112
- Investor invests in a group via `amgr deposit group`; see section 4.3.1.3, "Depositing Service Units to Group", on page 121.
- Investor withdraws from a group via `amgr withdraw group`; see section 4.3.3.3, "Withdrawing Service Units from Group", on page 124.
- Administrator transfers between groups via `amgr transfer group`; see section 4.3.8.5, "Transferring Service Units for Investors and Group", on page 134.

The difference between being active and inactive is that an active group can participate in transactions, but an inactive group cannot.

1.6.2 Project, Project Account

A project account is designed to represent a project such as a workflow. It has the following:

- Name
- Credit balance which it can use to run jobs
- One or more associated groups
- One or more associated users
- One or more associated clusters representing PBS complexes
- Accounting policy; see section 1.7.4, “Accounting Policy”, on page 23
- Optional independent start date
- Optional independent end date
- Optional metadata
- State: active or inactive

A project acquires credit when a group manager deposits group credit to the project account. A project spends credit when a user associated with the project runs a job and charges the job to the project account. A job can be charged to a project account only when it is run by an associated user on an associated complex. Any user associated with the project can charge a job to the project account. A project can use multiple clusters.

When a group manager deposits group credit to a project account, that allocation is for a specific period and in a specific currency (service unit). Currency is usually defined as resource usage, such as CPU hours or GPU hours, but can be in dollars or other monetary units.

Operations:

- Administrator adds project via `amgr add project`; see section 4.2.1.3, “Adding a Project”, on page 81
- Administrator updates project via `amgr update project`; see section 4.2.3.3, “Updating Projects”, on page 93
- Project member lists project via `amgr ls project`; see section 4.2.2.3, “Listing Projects”, on page 86
- Administrator removes project via `amgr rm project`; see section 4.2.4.3, “Removing a Project”, on page 102. Note that if a project has any associated current or past jobs or transactions, removing it results only in making it inactive.
- Group manager deposits service units to project via `amgr deposit project`; see section 4.3.1.2, “Depositing Service Units to Project”, on page 120
- Group manager withdraws service units from project via `amgr withdraw project`; see section 4.3.3.2, “Withdrawing Service Units from Project”, on page 123
- Project member, investor, or manager checks credit balance of project via `amgr checkbalance project`; see section 4.3.2.5, “Checking Service Unit Balance for Project”, on page 122
- Group manager transfers service units to and from projects via `amgr transfer project`; see section 4.3.8.4, “Transferring Service Units for Project”, on page 134
- Administrator or teller reconciles service units for project via `amgr reconcile project`; see section 4.3.6.3, “Reconciling Service Units for Project”, on page 131
- Group manager applies limits to dynamic service units for project via `amgr limit project`; see section 4.2.6, “Applying Limits to Dynamic Service Units”, on page 116
- Administrator or teller acquires service units for project via `amgr acquire project`; see section 4.3.5.4, “Acquiring Service Units for Project”, on page 129
- Project member, investor, or manager prechecks service unit balance for project via `amgr precheck project`; see section 4.3.4.1, “Prechecking a User or Project”, on page 125
- Project member, investor, or manager gets reports on projects via `amgr report project`; see section 4.2.5.3, “Getting Project Reports”, on page 106

The difference between being active and inactive is that an active project can run jobs and participate in transactions, but an inactive project cannot.

Project start and end times are not periods and are independent of periods.

When you create a project, you must assign an accounting policy and at least one cluster.

1.6.2.1 Project Attributes

Projects have a metadata attribute, consisting of comma-separated key-value pairs, where the keys are undefined. The administrator can set, update, or remove metadata when creating or updating a project.

Example 1-1: Add metadata to project:

```
amgr update project -n project1 -m + Owner:"Owner1"
```

Example 1-2: See metadata by getting a project report in prepaid mode:

```
amgr report project -n project1
-------------------------------------------------------------------------------
name     | period   | serviceunit | opening_balance | ... | metadata            |
-------------------------------------------------------------------------------
project1 | 2022.feb | cpu_hrs     | 0.0             | ... | {'Owner': 'Owner1'} |
```

Example 1-3: See metadata by listing project:

```
amgr ls project -n project1 -l
project1
    account = project1
    metadata = {'Owner': 'Owner1'}
...
```

Example 1-4: Remove metadata:

```
amgr update project -n project1 -m - Owner
```

Example 1-5: Verify by listing project:

```
amgr ls project -n project1 -l
project1
    account = project1
    metadata = {}
...
```

1.6.3 User, User Account

Entity representing an individual user and their associated account, including its credit balance. This user is typically a job submitter with the *user* role, although an individual user can have any role. A user has the following:

- Name, which is the username
- Credit balance
- One or more associated groups
- One or more associated clusters representing PBS complexes
- Optionally, one or more projects with which the user is associated
- Role; see section 1.4, “Roles”, on page 7
- Accounting policy; see section 1.7.4, “Accounting Policy”, on page 23
- Is active or inactive

A user acquires credit when a group manager deposits group credit to the user account. A user spends credit when that user runs a job and charges the job to their account. A job can be charged to a user account only when it is run by that user on a complex associated with that user. A user can be assigned to zero or more projects. When a user runs a job, they can charge the job to an associated project account, or to their own account.

When a group manager deposits group credit to a user account, that allocation is for a specific period and in a specific currency (service unit). Currency is usually defined as resource usage, such as CPU hours or GPU hours, but can be in dollars or other monetary units.

A user with any role can have an individual user account.

Operations:

- Administrator adds user via `amgr add user`; see section 4.2.1.2, “Adding a User”, on page 80
- Administrator updates user via `amgr update user`; see section 4.2.3.2, “Updating Users”, on page 93
- Job submitter lists self, or manager lists user user via `amgr ls user`; see section 4.2.2.2, “Listing Users”, on page 85
- Administrator removes user via `amgr rm user`; see section 4.2.4.2, “Removing a User”, on page 101. Note that if a user has any associated current or past jobs or transactions, removing it results only in making it inactive.
- Group manager deposits service units to user via `amgr deposit user`; see section 4.3.1.1, “Deposit Service Units to User”, on page 119
- Group manager withdraws service units from user via `amgr withdraw user`; see section 4.3.3.1, “Withdrawing Service Units from User”, on page 123
- Job submitter checks own credit balance, or manager checks credit balance of user via `amgr checkbalance user`; see section 4.3.2.4, “Checking Service Unit Balance for User”, on page 122
- Administrator transfers service units to and from users via `amgr transfer user`; see section 4.3.8.3, “Transferring Service Units for User”, on page 133
- Group manager applies limits to dynamic service units for user via `amgr limit user`; see section 4.2.6, “Applying Limits to Dynamic Service Units”, on page 116
- Administrator or teller acquires service units for user via `amgr acquire user`; see section 4.3.5.3, “Acquiring Service Units for User”, on page 129
- Administrator or teller reconciles service units for user via `amgr reconcile user`; see section 4.3.6.2, “Reconciling Service Units for User”, on page 130
- Job submitter prechecks own service unit balance, or manager prechecks service unit balance for user via `amgr precheck user`; see section 4.3.4.1, “Prechecking a User or Project”, on page 125
- Job submitter prechecks whether own service unit balance is sufficient for specific jobs, or manager prechecks service unit balance for user jobs, via `amgr precheck jobs`; see section 4.3.4.2, “Prechecking Jobs”, on page 126
- Job submitter gets report on self, or manager gets reports on user via `amgr report user`; see section 4.2.5.2, “Getting User Reports”, on page 104

The difference between being active and inactive is that an active user can run jobs and participate in transactions, but an inactive user cannot.

1.6.3.1 Requirements for Adding Job Submitters

When you create a user, you must assign a role, an accounting policy, and at least one cluster.

Each user you add to Budgets should already have an entry in the password file, with a password set, and a home directory on the Linux system where Budgets is installed.

When you add a user who will run jobs, that user must already be able to run jobs in a PBS complex.

1.7 Accounting Tools

1.7.1 Periods, Allocation Periods, Billing Periods

A period, also called a billing period or an allocation period, is a defined period of time with fixed start and end dates.

When a group manager deposits service units to a project or user account, that allocation is deposited for a specific period. The allocation is available to the project or user for that period only, expires at the end of the period, and is reported against that period.

Group accounts do not have associated periods.

The Budgets administrator creates all periods. You can create a hierarchy of billing periods where the parent period encompasses the child periods. For example the parent period can be Year, and the child periods can be Quarter1, Quarter2, etc. There is no limit to the depth of the hierarchy.

Operations:

- Administrator adds period via `amgr add period`; see section 4.2.1.6, “Adding a Period”, on page 83
- Administrator updates period via `amgr update period`; see section 4.2.3.6, “Updating a Period”, on page 96. Note that if a period has any associated jobs or transactions, you cannot update it.
- Any user lists any period via `amgr ls period`; see section 4.2.2.6, “Listing Periods”, on page 88
- Administrator removes period via `amgr rm period`; see section 4.2.4.6, “Removing a Period”, on page 103. Note that if a period has any associated current or past jobs or transactions, you cannot remove it.

1.7.1.1 Caveats for Creating Periods

- Make sure that periods at the same level do not overlap. For example, if Quarter1 ends March 31st, make sure that Quarter2 does not begin sooner than April 1st.
- If you want to create periods with a parent-child relationship, you must create the parent period first. You cannot add a parent to an existing child. For example, if you want Year as the parent and Quarter1, Quarter2, etc., as children, create Year first.
- If you create child periods, make sure that they fit within the parent period.
- Make sure that your period hierarchy is finalized BEFORE doing any transactions or running any jobs; you cannot update or remove periods once transactions have been performed or jobs have started.

1.7.2 Service Units

Budgets has two types of service units:

- You use *standard service units* to track and bill for resource usage, such as dollars, CPU hours, or GPU hours; this type is *SU_STANDARD*
- You use *dynamic service units* to set quotas on externally-managed resources such as storage; this type is *SU_DYNAMIC*

You can define as many service units as you need, and you can define each one to be whatever you need. You can define and use different service units for billing at each PBS complex. A job, project, or user can consume multiple service units. The default type is *SU_STANDARD*.

Operations:

- Administrator adds service unit via `amgr add serviceunit`; see section 4.2.1.7, “Adding a Service Unit”, on page 84
- Administrator updates service unit via `amgr update serviceunit`; see section 4.2.3.7, “Updating a Service Unit”, on page 97. You can update the type of a service unit only when it has no associated transactions or value updates. Use this command to set a service unit active or inactive.
- Any user lists service unit via `amgr ls serviceunit`; see section 4.2.2.7, “Listing Service Units”, on page 88
- Administrator removes service unit via `amgr rm serviceunit`; see section 4.2.4.7, “Removing a Service Unit”, on page 103. Note that if a service unit has any associated current or past transactions, removing it results only in making it inactive.
- Administrator uses `cron` script to update dynamic service unit usage `via amgr update dynamicvalues`; see section 4.2.3.9, “Updating Dynamic Service Unit Usage”, on page 98.
- Administrator sets a limit on the service unit via `amgr limit {user | project}`; see section 4.2.6, “Applying Limits to Dynamic Service Units”, on page 116
- Administrator defines each service unit in the billing formulas; see section 3.2.3, “Define Billing Formulas”, on page 62

The difference between being active and inactive is that an active service unit can be used for transactions or for quota checks, but an inactive one cannot; it cannot be invested, transferred, consumed, etc.

1.7.2.1 Standard Service Units

A standard service unit can be a monetary unit such as dollars, or it can represent an internally-managed resource such as CPU hours or GPU hours. Standard service units can be treated like a currency.

The Budgets hook runs at the PBS complex where a job runs and tracks standard service unit usage by the job. Budgets debits the account of the job owner for the usage by the job. When you manage a PBS complex via Budgets, each PBS job can run only when the job can be charged to a project or user account that has sufficient standard service units.

1.7.2.1.i Adding and Removing Standard Service Units

You can create separate standard service units for each cluster.

To create a standard service unit and add it to Budgets:

1. Define the service unit in a billing formula; see section 3.2.3, “Define Billing Formulas”, on page 62
2. Add the service unit to Budgets via amgr `add serviceunit -n <name of new service unit>`; see section 4.2.1.7, “Adding a Service Unit”, on page 84

To remove a standard service unit:

1. Remove the service unit from all formulas, otherwise jobs will not run
2. Remove the service unit from Budgets via `amgr rm serviceunit -n <service unit name>`; see section 4.2.4.7, “Removing a Service Unit”, on page 103

1.7.2.2 Dynamic Service Units

A dynamic service unit tracks an external resource such as storage. You use a dynamic service unit by setting a limit on it to establish a quota. You can set separate limits on a dynamic service unit for each user and project, and for each user or project, you can specify a different limit for each period, via `amgr limit {user | project}`; see section 4.2.6, “Applying Limits to Dynamic Service Units”, on page 116. To set a limit on a dynamic service unit for a user or project, you must be the group manager for the group that provides standard service units for that user or project.

Budgets checks whether a job owner has hit a quota before starting each job. If a job owner hits a quota, they cannot start any more jobs until either the limit is raised or usage goes down.

1.7.2.2.i Caveats for Dynamic Service Units

If a dynamic service unit has no limit set on it, the limit is zero. If there are active dynamic service units, all jobs are checked against those quotas, and a zero quota will stop any job from running. Make sure that you don't unintentionally stop non-target users or projects from running jobs:

- Make sure you set the quota for all users and projects
- When you specify the period, make sure you either:
  - Set the desired quota at the top level of the period hierarchy
  - Set a very high quota at the top level of the period hierarchy, and a more restrictive quota for the period you need to control

1.7.2.2.ii Attributes for Dynamic Service Units

The data_lifetime Budgets configuration attribute specifies the maximum time period between updates to dynamic service units. The default value is 3600 seconds. See section 3.3, "Setting Budgets Configuration Attributes", on page 73.

1.7.2.2.iii Adding and Removing Dynamic Service Units

To create and use a dynamic service unit, you add it to Budgets, set a limit, and keep its value updated:

1. Add the service unit to Budgets via amgr `add serviceunit -n <name of new service unit>`; see section 4.2.1.7, "Adding a Service Unit", on page 84
2. Set a limit on the service unit via `amgr limit {user | project}`; see section 4.2.6, "Applying Limits to Dynamic Service Units", on page 116
3. Periodically update the value of the service unit by having a `cron` script call `amgr update dynamicvalues`; see section 4.2.3.9, "Updating Dynamic Service Unit Usage", on page 98. The script should update the value at intervals that are smaller than the limit set in the data_lifetime attribute; the default value is 3600 seconds. See section 3.3, "Setting Budgets Configuration Attributes", on page 73

To remove a dynamic service unit from Budgets, use `amgr rm serviceunit -n <service unit name>`; see section 4.2.4.7, "Removing a Service Unit", on page 103

1.7.2.2.iv Checking Quotas (Limits on Dynamic Service Units)

There may be quotas set on externally-managed resources such as storage. A quota is a limit on a dynamic service unit. To see quotas, list all service units:

```
amgr ls serviceunit
```

See section 4.2.2.7, "Listing Service Units", on page 88.

1.7.2.3 Examples of Storage Quotas via Dynamic Service Units

Example 1-6: Setting user and project quotas:

Set user quota:

```
amgr limit user -n user1 -s storage 12.0 -p 2022
```

Set project quota:

```
amgr limit project -n project1 -s storage 25.0 -p 2022
```

Example 1-7: Setting consumption by user and project. This call is typically made via a `cron` job, and you would want to make sure you set this more often than the limit in the data_lifetime attribute:

Set consumption for user1:

```
amgr update dynamicvalues -v '{"storage": {"user1": {"total":8}}}'
```

Set consumption for project1:

```
amgr update dynamicvalues -v '{"storage": {"project1": {"total":30}}}'
```

Example 1-8: Reporting storage usage:

Report user storage data:

```
amgr report user -n user1 -t SU_DYNAMIC
------------------------------------------------------------------------------------
name     | serviceunit  | period    | limit   | last_reported_time  | total_consumed
------------------------------------------------------------------------------------
user1   | storage      | 2022.feb  | 12.0    | 2022-02-11 18:58... | 8.0
```

Report project storage data:

```
amgr report project -n project1 -t SU_DYNAMIC
------------------------------------------------------------------------------------
name     | serviceunit  | period    | limit   | last_reported_time  | total_consumed
------------------------------------------------------------------------------------
project1 | storage      | 2022.feb  | 25.0    | 2022-02-11 18:58... | 30.0
```

Example 1-9: Queued job for project1 won't run. Why? Because it is over quota as shown in the report above.

```
qstat 3269
Job id                 Name             User              Time Use S Queue
---------------------  ---------------- ----------------  -------- - -----
3269.testbed           workq_test       user1                    0 Q workq
qstat -f 3269 | grep comment
comment = Not Running: PBS Error: Budgets: Consumption has reached the
limit for a dynamic service unit storage
```

1.7.2.4 Rules for Using Service Units

- You can change the type of a service unit, but there are restrictions:
  - You can change standard to dynamic only when no transactions have taken place for that service unit
  - You can change dynamic to standard only when no updates have been made to the usage of that service unit
- In the billing formula file, you can use only standard service units.
- All active dynamic service units must have a limit set in order for jobs to run
- When you create a new child period, it inherits its limits for any dynamic service units from its parent
- If you apply a limit directly to a dynamic service unit for a period, that overrides any inherited limit
- If you apply a limit to a period, all child periods that are not directly limited inherit the limit; similarly, limits on child periods are inherited by children of those child periods, when no direct limits have been set
- Only administrators can update dynamicvalues
- The maximum amount of each service unit an account can hold is 999999999999.99.

1.7.3 Transactions

A transaction is an operation on a service unit, such as a deposit or transfer. We say a transaction is an element in Budgets. Budgets also uses checks on account balances; these are not transactions, but they are also elements in Budgets.

Table 1-1: Transactions and Checks

Name	Operation	Format	When	Purpose
Deposit	Deposit service units to user, project, or group account Depositing Service Units	Unique float	Any time	Investor funds group. Group manager funds user or project from group account.
Withdraw	Withdraw service units from user, project, or group account Withdrawing Service Units	Unique float	Any time	Investor withdraws funds from group. Group manager withdraws service units from user or project and returns them to group account.
Transfer	Move service units from one investor or user, project, or group account to another Transferring Service Units	Unique float	Any time	Administrator transfers service units between investors, groups, periods, projects, and users. Useful when period expires leaving unused funds; these funds can be transferred where needed.
Refund	Refund service units from group account to user or project account Refunding Service Units	Job ID plus date, time stamp, and operation (shown in report)	Any time	Administrator refunds job owner for costs out of job owner's control, for example re-run caused by node failure. Refunds are strictly per-job.
Acquire	Service units from job owner's account are put in escrow before running the job Acquiring Service Units	Job ID plus date, time stamp, and operation (shown in report)	Before job runs	Hook moves credit from job owner's account to escrow. Can be used as a debug tool by administrator.
Reconcile	Service units consumed by job are removed from escrow. Returns unused service units in escrow, or if job consumed more than requested, debits more from job owner account Reconciling Service Units	Job ID plus date, time stamp, and operation (shown in report)	After job runs	Hook reconciles accounts after job ends: debits escrow for amount consumed by job and returns unused credit to job owner, or if job consumed more than requested, debits more from the job owner's account. Can be used as a debug tool by administrator.
Precheck	Check job owner account for sufficient service units; disallow queueing job if insufficient Prechecking Service Unit Balance	Not a transaction; no ID	Before queueing job	Hook can optionally disallow queueing of jobs owned by accounts with insufficient service units. Can be used as a debug tool by administrator.
Checkbalance	Check job owner account for sufficient service units; do not run job if insufficient Checking Service Unit Balance	Not a transaction; no ID	Before job runs	Hook checks job owner credit balance before allowing job to run.

You can get reports on transactions; see section 4.2.5.5, “Getting Job and Transaction Reports”, on page 115.

1.7.3.1 Transaction IDs

Every transaction has a transaction ID.

- Transactions associated with jobs, such as acquiring service units to run the job, consist of the job ID, a date, a timestamp, and an operation, shown in the report. For example:

  ```
  1235.myserver   2022-10-06  17:15:23.760451  ...  acquired  ...
  ```

- Other transactions have a unique floating-point number for their transaction ID. This format has 10 digits before the point and 7 after the point, for example:

  ```
  0123456789.1234567
  ```

1.7.4 Accounting Policy

You must specify an accounting policy for each user and project when you add them to Budgets. The policy determines how the entity is charged for its jobs. You can set the accounting policy when you add or update the entity via the `-A <accounting policy>` option to the `amgr add` or `amgr update` commands. There is no default policy. The accounting policy is case-sensitive, and it is lowercase. You have the following options:

begin_period

The user or project account is charged when the job begins.

end_period

The user or project account is charged when the job ends.

proportionate

The user or project account is charged during all periods when the job runs, and each period is charged in proportion to the usage during that period.

1.7.5 Clusters

A cluster is a data structure representing a PBS complex. Each cluster is named for its PBS server; the server name is the value of the PBS_SERVER parameter in the `/etc/pbs.conf` file. Cluster names (and therefore PBS server names) must be unique. The cluster formulas must be the same as the formulas at the complex, otherwise jobs won't run. If you log into a PBS complex and change a formula there, update the cluster data structure at the Budgets host via `amgr update cluster`.

When you run `amgr add cluster` or `amgr update cluster`, you are also updating the database with the formulas for the cluster.

A cluster has the following:

- Name; this must be the name of the PBS server for the PBS complex, found in the PBS_SERVER parameter in the `/etc/pbs.conf` file
- Billing formulas; see section 3.2.3, “Define Billing Formulas”, on page 62
- Is active or inactive

In order for a user or project to be able to run a job at a PBS complex, its representative cluster must be associated with the user or project.

Cluster operations:

- Administrator adds a cluster to Budgets via `amgr add cluster`; see section 4.2.1.5, "Adding a Cluster", on page 83
- Administrator updates a cluster via `amgr update cluster`; see section 4.2.3.5, "Updating Clusters", on page 96
- Any user lists a cluster via `amgr ls cluster`; see section 4.2.2.5, "Listing Clusters", on page 87
- Administrator removes a cluster via `amgr rm cluster`; see section 4.2.4.5, "Removing a Cluster", on page 102. If the cluster has any associated jobs or transactions, the remove operation only makes the cluster inactive.

The difference between an active and an inactive cluster is that an active cluster can run jobs and participate in transactions, but an inactive cluster cannot. Otherwise they are the same.

Figure 1-3 shows the relationship between a cluster and its associated PBS complex.

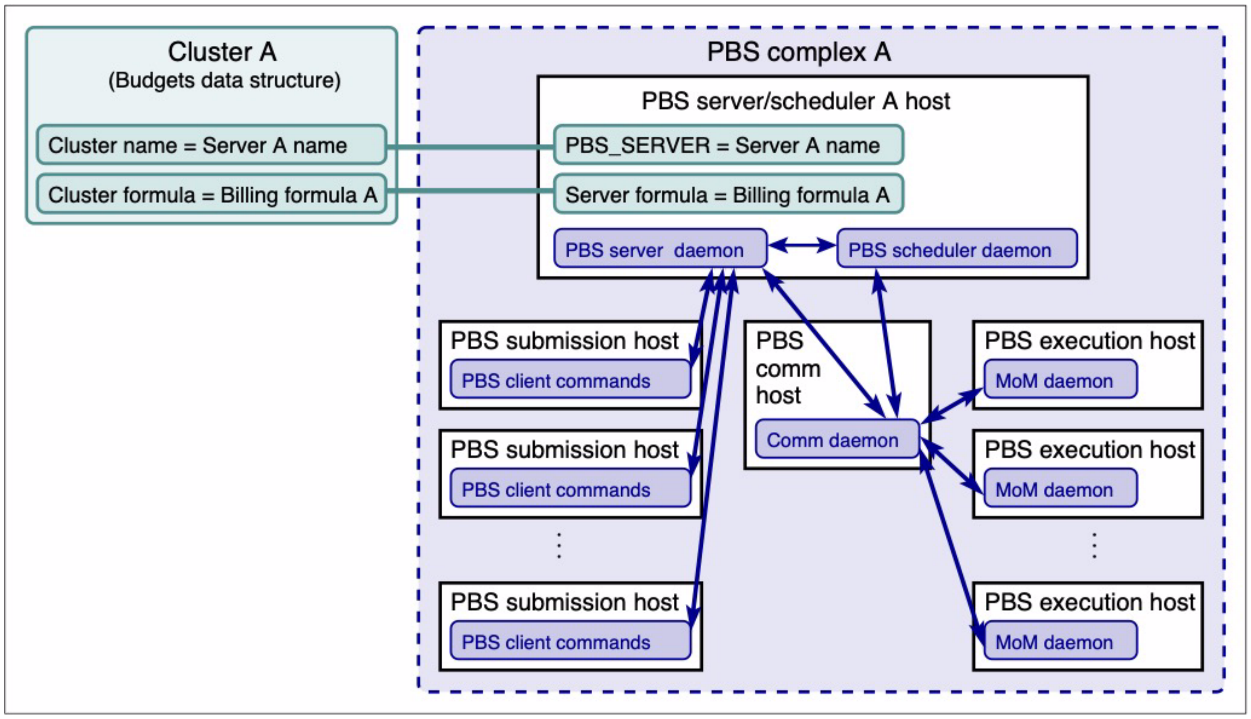

Figure 1-3: Relationship between cluster and PBS complex

1.8 Summary of Setting Budgets Up for Postpaid or Prepaid Mode

1.8.1 Summary of Using Postpaid Mode

To set Budgets up to use postpaid mode, you need to create the relevant clusters, an active period, the desired service unit, and accounts for any job submitters. If you want to use projects, you need to create any relevant project accounts. You also need to make sure that each job submitter is a member of the relevant cluster.

If job submitters want to pay their balance off early, you need to create a group for the job submitters, a manager for the group, an investor for the group, an investment to the group, and a grant from the group to each job submitter.

1.8.2 Summary of Using Prepaid Mode

To set Budgets up to use prepaid mode, you need to create the relevant clusters, an active period, the desired service unit, and accounts for any job submitters. You also need to create a group for the job submitters, a manager for the group, an investor for the group, an investment to the group, and a grant from the group to each job submitter. You also need to make sure that each job submitter is a member of the relevant cluster.

If you want to use projects, you need to create any relevant project accounts, and add the submitters as members of the projects.

1.9 Caveats and Restrictions

- The Budgets administrator account must exist and have the *admin* role at all times. Make sure you never remove the *admin* role from your administrator account. If you have no administrator with *admin* role, you cannot use Budgets. If you do not have another administrator account with the *admin* role, do not disable your administrator account as shown in section 2.11, "Changing Budgets Administrator to New Username", on page 58, step 3, Permanently disable pbsadmin.
- Soft walltime is not supported.
- Shrink-to-fit jobs are not supported.
- If the license server is not reachable, jobs will continue to run for up to 3 hours. After that, jobs cannot start or be reconciled.
- In postpaid mode, a job that is deleted via `qdel -Wforce` is not billed, because it has no *E* record.
- If, in postpaid mode, two groups invest in the same user or project account, but the first group's investment serves only to bring the account from a negative value up to zero, the second group is given full ownership if you switch to prepaid mode at a time that is not on a period boundary. You can avoid this by always switching between modes on a period boundary.

1.10 Troubleshooting

1.10.1 Using Logfiles for Troubleshooting

1.10.1.1 Using Budgets Logfile

Look in the Budgets server log for information about Budgets. Budgets writes its logfile to `/var/spool/am/<Budgets server hostname>_<value of AM_PORT>_server.log`.

Each day, Budgets automatically renames old log files to "<original filename>-<year>-<month>-<day of month>". So if Budgets writes a log file named "/var/spool/am/myserverhost_8000_server.log", and renames it on the 14th of March 2022, the new name is "/var/spool/am/myserverhost_8000_server.log-2022-march-14".

This file lists the requests made to the Budgets server and the resulting actions, and information about problems. Some typical problems and their indicators:

- License has expired: logfile shows "license unavailable"
- A cluster name is different from its associated PBS server : logfile shows "could not resolve hostname <cluster name>"
- A request from an unauthenticated user: logfile shows "unable to login. Username or password is incorrect'
- A user tries to use a project they're not a member of : logfile shows "user <username> is not authorized for <project name> project"

1.10.1.2 Using PBS Server Logfile

Look in the PBS server log for logging by the Budgets hooks am_hook and am_hook_periodic. The default location for PBS server logs is `$PBS_HOME/server_logs`. See "Event Logging" on page 428 in the PBS Professional Administrator's Guide.

1.10.2 Symptoms

- Symptoms of losing connection with the AMS module:
  - If users can no longer log into or out of Budgets, the connection with the AMS module may have been lost.
  - If Budgets stops jobs from running, check the Budgets logfile or a job comment. If you see Budgets complaining about "connection refused", that means Budgets can't reach the AMS module.
  - If you reinstall the AMS module, that breaks its connection with Budgets.

  To reestablish the connection, run this script as root:

  `/opt/am/libexec/am_auth_register`
- Symptom of insufficient credit:
  - A job returns an error saying there is not enough credit

1.11 Formats in Budgets

1.11.1 Name Formats

- Name format for projects, groups, periods:

  Allowed characters: a-z,A-Z,0-9,_,-,.,$

  Max length: 256 characters
- Name format for service units:

  Allowed characters: a-z,A-Z,0-9,_,$

  Max length: 30 characters
- Name format for usernames is OS-dependent

2 Installing and Upgrading Budgets

2.1 Supported Platforms

2.1.1 OpenSSL Requirement

PBS requires OpenSSL 1.1.1. If this is not already present on your platform, you must install it.

2.1.2 PBS Components

PBS Professional is made up of the following components:

- PBS Professional server/scheduler daemon on PBS Professional server/scheduler host/head node
- PBS Professional MoM daemon on execution host/compute node, with the following options:
    - On premise
    - Burst in cloud via PBS Cloud (optional)
- PBS Professional client commands on PBS submission host/client host
- PBS Professional communication daemon on communication host
- PBS Cloud module on service node (where AMS module runs) (optional)
- Budgets server on Budgets head node (optional)
- Budgets AMS module on service node (where PBS Cloud module runs) (optional)
- Budgets client commands on Budgets client host (optional)
- Simulate module:
    - When using PBS Cloud, Simulate must be installed on PBS Professional server/scheduler host
    - When not using PBS Cloud, Simulate can be installed on any supported host

2.1.3 Supported Platforms for PBS Components

PBS components are supported on the following platforms. A **(d)** indicates that support is deprecated:

Table 2-1: Supported Platforms

Maker	Version	Chip set	Components							
			PBS Professional				Cloud + AMS		Budgets	Simulate
			Server Sched	MoM on prem	Comm	Client cmds	Cloud module + AMS	MoM burst node	Head node + client cmds	Head node
CentOS	7	x86_64	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
	7	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	No
Red Hat Enterprise Linux RHEL	7	x86_64	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	7	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	Yes
	7 MLS	x86_64	Yes	Yes	Yes	Yes	No	No	No	No
	8	x86_64	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	8	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	Yes
Rocky Linux	8	x86_64	Yes	Yes	Yes	Yes	No	Yes	No	No
	8	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	No
SUSE SLES	12	x86_64	Yes	Yes	Yes	Yes	Yes *	Yes	Yes	Yes
	12	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	No
	15	x86_64	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
	15	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	Yes
Ubuntu	18.04	x86_64	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
	18.04	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	Yes
	20.04	x86_64	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
	20.04	ARM64	Yes	Yes	Yes	Yes	No	Yes	No	Yes
HPE Cray Shasta	1.1 SLES 15	x86_64	Yes	Yes	Yes	Yes	Yes *	No	Yes *	Yes
	1.1 RHEL 7	x86_64	Yes	Yes	Yes	Yes	No	No	No	Yes
NEC SX-Aurora TSUBASA			Yes	Yes	Yes	Yes	No	No	No	Yes
Windows	10 Pro	x86_64	No	Yes	No	Yes	No	Yes	No	No

Table 2-1: Supported Platforms

Maker	Version	Chip set	Components							
			PBS Professional				Cloud + AMS		Budgets	Simulate
			Server Sched	MoM on prem	Comm	Client cmds	Cloud module + AMS	MoM burst node	Head node + client cmds	Head node
	11 Pro	x86_64	No	Yes	No	Yes	No	Yes	No	No
	Server 2016	x86_64	No	Yes	No	Yes	No	Yes	No	No
	Server 2019	x86_64	No	Yes	No	Yes	No	Yes	No	No

2.1.3.1 * SLES Restrictions for Cloud and Budgets Nodes

The following restrictions apply when using SLES on service node host for PBS Cloud or head node host for Budgets:

- Each SLES host must be registered with the SUSE Customer Center via SUSEConnect, and have a support contract. This happens automatically for cloud nodes.
- SLES hosts require Docker Enterprise Edition.

2.1.4 Supported Platforms for Nodes Burst in Cloud

- Linux: any Linux platform that supports both PBS MoM and `cloud-init`
- Windows: 10, Server 2012

All versions of `cloud-init` are supported.

2.1.5 Restrictions on Simulate Module Location when Using PBS Cloud

If you will use the PBS Cloud module, you must install Simulate on the PBS Professional server/scheduler host (the PBS Professional head node).

2.1.6 Hosts for Budgets Client Commands

Any host where you install just the client commands must be able to reach the Budgets server, and must be a supported platform for the Budgets client commands.

2.2 Recommended Configurations

Budgets is typically configured with a *head node* (where the PBS Professional server and the Budgets server run) and a *service node* (where PBS Cloud and the AMS module run). We prefer not to run Docker on a heavily loaded PBS server host, so we typically put the elements requiring Docker on the service node.

Head node and service node can be one of either:

- Both on premises
- Both in cloud

Do not put one on premises and one in the cloud.

There are no restrictions on client command-only hosts.

2.2.1 Installation Directory

The default location for the Budgets server is `/var/spool/am`. You can install the Budgets server in the default location or a non-default location of your choice. The AMS module is always installed in the same location; this location is controlled by the AMS installer.

2.2.2 Recommended Configuration for Larger Installations

For larger installations using on premises hosts with optional cloud bursting:

- Head, service, and first N execution nodes are on premises:
  - On head node, PBS server daemon, PBS scheduler daemon, and Budgets server
  - On service node, AMS module running in container, and PBS Cloud running in separate container
  - Execution hosts running MoM daemons
- Cloud nodes for extra execution nodes
- VPN connection to the cloud you will use
- Client commands go on any Linux host
- All components are mix-and-match (with Docker restriction)

2.2.3 Recommended Configuration for Smaller Installations

For smaller installations and cloud-only installations where the workload is low enough:

- All components can be hosted in the cloud
- All components can run on the same node
- You can run Docker on the same node as the PBS server/scheduler
- You can also run a cloud head node and separate cloud service node:
  - Cloud head node, running PBS Professional server/scheduler and Budgets server
  - Cloud service node, running AMS module in container, and PBS Cloud in separate container
- Client commands go on any Linux host, but a user must be able to reach the Budgets port on the cloud host
- No VPN is required for this configuration

2.3 Whether or Not to Start with Failover

Consider configuring Budgets for failover. Without failover, the `AM_HOME` directory is on a local drive, but with failover, it's on a separate host. We recommend doing a fresh install for failover, which means starting with an empty database. For failover configuration instructions, see section 2.9, “Configuring Budgets for Failover”, on page 53.

2.4 Prerequisites

2.4.1 Altair Software Components

- Budgets server
- Budgets client commands
- Budgets authentication module (AMS)
- PBS Professional 2022.1.0 or later, installed and running
  - PBS Professional server/scheduler(s)/comm(s)
  - PBS Professional MoM(s)
  - PBS Professional client commands
- Altair License Manager 14.5.1 or newer, installed and running
- PBSProNodes 20.0 license features

2.4.2 Third-party Software Components

- docker-ce v19.x or later for most systems
- docker-ee v19.x or later for SLES
- python3
- python3-pip
- openssl

2.4.3 Job Requirements

Each job must request the compute resources that are used in the billing formulas used at that complex. For the default formula, this means walltime and ncpus. Make sure that every PBS job requests ncpus and walltime when it runs. Each job can have these set at submission by the job submitter or later via `qalter`, can inherit a value from the server or queue, or can be assigned a value by a hook. For ncpus, the server attribute default_chunk.ncpus may take care of the requirement.

Job walltime:

- If a job's walltime is extended, Budgets takes that into account.
- Soft walltime is not supported.
- Shrink-to-fit jobs are not supported.

2.5 Installation Steps for All Locations

Follow the steps in this section no matter where you are installing PBS Cloud.

Follow the steps in the order they are presented: when it comes up, the Budgets server needs use the passwords and certificates to communicate with the AMS module, and needs to verify the various accounts.

2.5.1 Create Required User Accounts

2.5.1.1 Budgets Administrator

Budgets requires an administrator. The administrator configures Budgets.

We recommend that the username for the administrator account be *pbsadmin* (same account used for PBS administrator). You can use any username for the Budgets administrator. Where you see "pbsadmin" in the instructions, substitute the actual administrator username.

You can install and configure Budgets using pbsadmin for the administrator username, then switch to another username later. See section 2.11, “Changing Budgets Administrator to New Username”, on page 58.

2.5.1.1.i Requirements for Administrator Account

- This account must exist and have the *admin* role at all times. Make sure you never remove the *admin* role from your administrator account. If you have no administrator with *admin* role, you cannot use Budgets.
- The administrator account should not be used for the teller.
- The home directory for the administrator should exist on the PBS server host, and on the Budgets host, if it is a separate machine.
- Administrator account must have an account on the PBS server host, and on the Budgets host, if it is a separate machine.
- Administrator should be able to `ssh` to the Budgets server host without a password. Administrator must have passwordless `ssh` set up from the PBS server host to the Budgets server host. We cover this in section 2.5.3, “Set Up Passwordless SSH for Administrator and Teller”, on page 35.

2.5.1.2 Teller

Budgets requires a teller user to process its service unit transactions. When the Budgets hook performs transactions, it does so as the teller.

2.5.1.2.i Requirements for Teller Account

- We recommend that the username for the teller account be *amteller*
- Teller should be a dedicated account used only for the *teller* role
- Teller account must have an account on the PBS server host, and on the Budgets host, if it is a separate machine.
- Teller should be able to `ssh` to the Budgets server host without a password. Teller must have passwordless `ssh` set up from the PBS server host to the Budgets server host. We cover this in section 2.5.3, “Set Up Passwordless SSH for Administrator and Teller”, on page 35.
- The teller account should not be used for pbsadmin
- The teller account does not need to be root or administrator

2.5.1.3 Database User

Budgets requires a database user for its database.

2.5.1.3.i Requirements for Database User Account

- We recommend that the username for the database user account be *pbsdata*.
- The pbsdata account should have an ID <1000 so that any processes which run under this user are protected from the Out Of Memory (OOM) killer and run with the correct level of privilege.

2.5.1.4 Job Submitters

Users who submit jobs to PBS Professional have to exist on the system where Budgets is installed, but these users can be added to Budgets after installation.

2.5.1.4.i Requirements for Job Submitters

- Job submitters must already be able to run jobs in a PBS complex.
- Each user you add to Budgets should already have an entry in the password file, with a password set, and a home directory on the Linux system where Budgets is installed.

2.5.1.5 Configuring Required Accounts for Budgets

- Add pbsadmin as the Budgets administrator, and set a password:
  ```
  adduser -u 901 pbsadmin
  passwd pbsadmin <password>
  ```
- Add amteller as the Budgets teller, and set a password:
  ```
  adduser amteller
  passwd amteller <password>
  ```
- Add pbsdata as the Budgets database user, and set a password:
  ```
  adduser -u 900 pbsdata
  passwd pbsdata <password>
  ```

2.5.2 Allow Interaction with PBS Professional

In order to work on hooks, you have to be root, so we add specific actions to the `sudoers` file on the Budgets and PBS server hosts for pbsadmin to work as root when using `amgr` and `qmgr` to import and export hook files.

If you will install Budgets in a non-default location, make sure that `AM_EXEC` is set correctly.

2.5.2.1 Budgets Server and PBS Server on Same Host

On the Budgets/PBS server host, edit `/etc/sudoers`, and add the following lines. Replace the $AM_EXEC and $PBS_EXEC variables with the actual paths.

- Allow Budgets administrator to act as root for Budgets commands:
  ```
  Cmnd_Alias AM_SERVER_CMD = $AM_EXEC/python/bin/python3 $AM_EXEC/hooks/pbs/pbs_set_formula.py*
  ```
- Allow Budgets administrator to set the formulas as root:
  ```
  Defaults!AM_SERVER_CMD !requiretty
  ```
- Allow teller to get a security token:
  ```
  Cmnd_Alias AM_CLIENT_CMD = $AM_EXEC/python/bin/amgr sshlogin
  ```
- Allow `amgr` command to update hook formulas:
  ```
  Cmnd_Alias BUDGETS_IMPORTS = $PBS_EXEC/bin/qmgr -c i h am_hook application/x-config default
      .am/tmp*, $PBS_EXEC/bin/qmgr -c i h am_hook_periodic application/x-config default .am/tmp*,
      $PBS_EXEC/bin/qmgr -c export hook am_hook application/x-config default
  ```
- Allow the Budgets administrator account to act as root for the `qmgr` commands listed above:
  ```
  <budget administrator> ALL=(root) NOPASSWD: BUDGETS_IMPORTS
  ```
- Allow hooks to work in the background with no `tty` interface:
  ```
  Defaults!AM_CLIENT_CMD !requiretty
  ```
- Allow hooks to work in the background with no `tty` interface:
  ```
  Defaults!BUDGETS_IMPORTS !requiretty
  ```

2.5.2.2 Budgets Server and PBS Server on Separate Hosts

2.5.2.2.i Changes to sudoers on Budgets Server Host

On the Budgets server host, edit `/etc/sudoers`, and add the following lines. Replace the $AM_EXEC and $PBS_EXEC variables with the actual paths.

- Allow Budgets administrator to act as root for the `qmgr` commands listed above:
  ```
  Cmnd_Alias AM_SERVER_CMD = $AM _EXEC/python/bin/python3 $AM_EXEC/hooks/pbs/pbs_set_formula.py*
  ```
- Allow the Budgets administrator to set the formulas as root:
  ```
  Defaults!AM_SERVER_CMD !requiretty
  ```

2.5.2.2.ii Changes to sudoers on PBS Server Host

On the PBS server host, edit `/etc/sudoers`, and add the following lines. Replace the $AM_EXEC and $PBS_EXEC variables with the actual paths. We give an example showing default paths below:

- Allow teller to get a security token:
  ```
  Cmnd_Alias AM_CLIENT_CMD = $AM_EXEC/python/bin/amgr sshlogin
  ```
- Allow `amgr` command to update hook formulas:
  ```
  Cmnd_Alias BUDGETS_IMPORTS = $PBS_EXEC/bin/qmgr -c i h am_hook application/x-config default
      .am/tmp*, $PBS_EXEC/bin/qmgr -c i h am_hook_periodic application/x-config default .am/tmp*,
      $PBS_EXEC/bin/qmgr -c export hook am_hook application/x-config default
  ```
- Allow the Budgets administrator account to act as root for the `qmgr` commands listed above:
  ```
  <budget administrator> ALL=(root) NOPASSWD: BUDGETS_IMPORTS
  ```
- Allow hooks to work in the background with no `tty` interface:
  ```
  Defaults!AM_CLIENT_CMD !requiretty
  ```
- Allow hooks to work in the background with no `tty` interface:
  ```
  Defaults!BUDGETS_IMPORTS !requiretty
  ```

2.5.3 Set Up Passwordless SSH for Administrator and Teller

The administrator needs to have passwordless `ssh` available from the Budgets server host to the PBS server host. The teller needs to have passwordless `ssh` available from the PBS server host to the Budgets server host.

2.5.3.1 Setting Up Passwordless SSH for Administrator

To give the administrator passwordless `ssh` from the Budgets server host to the PBS server host, generate a public authentication key at the Budgets server host and append it to the `~/.ssh/authorized_keys` file at the PBS server host. Here are the steps:

1. Log in to the Budgets server host as the administrator
2. Check for an existing SSH key pair:
   ```
   ls -al ~/.ssh/id_*.pub
   ```
3. If you find existing keys, you can use those, or you can back up the old keys and generate a new pair. If you don't find existing keys, generate a new SSH key pair:
   ```
   ssh-keygen
   ```
4. Copy the contents of `id_rsa.pub`
5. Log in to the PBS server as the administrator
6. In the home directory, check for the .ssh directory. If it does not exist, create it:
   ```
   mkdir -p .ssh
   cd .ssh/
   ```
7. Create the `authorized_keys` file in the .ssh directory:
   a. Paste the contents of `id_rsa.pub` that you copied from the Budgets server host
   b. Save the file as "authorized_keys"
8. Change the permission of `authorized_keys` to *600*:
   ```
   chmod 600 authorized_keys
   ```

2.5.3.2 Setting Up Passwordless SSH for Teller

To give the teller passwordless `ssh` from the PBS server host to the Budgets server host, generate a public authentication key at the PBS server host and append it to the `~/.ssh/authorized_keys` file at the Budgets server host. Here are the steps:

1. Log in to the PBS server host as the teller
2. Check for an existing SSH key pair:
   ```
   ls -al ~/.ssh/id_*.pub
   ```
3. If you find existing keys, you can use those, or you can back up the old keys and generate a new pair. If you don't find existing keys, generate a new SSH key pair:
   ```
   ssh-keygen
   ```
4. Copy the contents of `id_rsa.pub`
5. Log in to the Budgets server as the teller
6. In the home directory, check for the `.ssh` directory. If it does not exist, create it:
   ```
   mkdir -p .ssh
   cd .ssh/
   ```
7. Create the `authorized_keys` file in the `.ssh` directory:
   a. Paste the contents of `id_rsa.pub` that you copied from the PBS server host
   b. Save the file as "authorized_keys"
8. Change the permission of `authorized_keys` to *600*:
   ```
   chmod 600 authorized_keys
   ```

2.5.4 Set Budgets Paths

Append executable path to PATH environment variable for all users of Budgets (administrator, investors, managers, teller, job submitters). You can set this in `/etc/bashrc`:

```
export PATH=$PATH:$AM_EXEC/python/bin
```

For the default path, this is:

```
export PATH=$PATH:/opt/am/python/bin
```

2.6 Installation Steps for Default Location

2.6.1 Install Utilities and Docker

Install utilities and `docker` on the service node. The directory is not important.

- For CentOS or RedHat:

  Log in as root to the service node (the machine where the AMS module is to be installed).

  ```
  yum install -y yum-utils
  yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
  yum install docker-ce docker-ce-cli containerd.io
  systemctl enable docker
  systemctl start docker
  yum install python3 python3-pip
  yum install openssl
  ```

- For SLES12 or 15:

  Log in as root to the service node (the machine where the AMS module is to be installed).

  For SLES 12:

  ```
      sudo SUSEConnect -p sle-module-containers/12/x86_64 -r ''
  ```

  For SLES 15:

  ```
      sudo SUSEConnect -p sle-module-containers/15.1/x86_64 -r ''
  sudo zypper install docker
  sudo systemctl enable docker.service
  sudo systemctl start docker.service
  ```

  Configure the firewall to allow forwarding of Docker traffic to the external network:

  ```
  Set FW_ROUTE="yes" in /etc/sysconfig/SuSEfirewall2
  zypper install python3-pip
  ```

- For Ubuntu:

  Log in as root to the service node (the machine where the AMS module is to be installed).

  ```
  sudo apt-get update
  sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-proper-
      ties-common
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
  sudo apt-key fingerprint 0EBFCD88
  ```

  The key should match the second line in the output; validate the last 8 characters. Example of second line:

  9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

  ```
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
      $(lsb_release -cs) stable"
  sudo apt-get update
  sudo apt-get install docker-ce docker-ce-cli containerd.io
  sudo apt-get install python3-pip
  ```

  This can install a number of required dependencies, and may take a few minutes.

  ```
  sudo apt-get install openssl
  sudo systemctl enable docker.service
  ```

```
sudo systemctl start docker.service
```

2.6.2 Download Budgets Server and AMS Modules

1. Log in as root to the Budgets server host
2. Go to the default installation location:

   ```
   # cd /var/spool
   ```

3. On the Budgets server host, download the main file containing both the Budgets server and the AMS module. The filename has the following format:

   PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

   For example:

   PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.gz
4. Untar the main file:

   tar xvfz PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

   For example:

   ```
   tar xvfz PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.gz
   ```

   This creates the following:

   - File named "ams-installer.tar.gz" containing the AMS installer
   - Directory named "am" containing the Budgets server module
5. Copy `ams_installer.tar.gz` over to the service node

2.6.3 Install AMS Module on Service Node

Install the AMS module on the service node. You can start in any directory.

1. Untar the AMS installer file:

   `Tar xvfz ams-installer.tar.gz`

   This gives a directory named "ams-installer", containing:

   - A file named "README.md "
   - A package named "ams-installer.zip"

2. Unzip `ams-installer.zip`:

   `unzip ams-installer.zip`

   This gives a directory named "ams-installation", containing:

   - Directory named "packages" containing the AMS package of folders and configuration files
   - Directory named "pbsworks-packager" containing the Python module that installs AMS

3. Change to `ams-installation` directory:

   `cd ams-installation`

4. Use the Python module to install AMS:

   `python3 -m pip install --upgrade --ignore-installed pbsworks-packager/`
   `/usr/local/bin/pkgr` (Please stay in the AMS installer directory for this step)

5. Answer the questions in the dialogue:

   a. Choose option: 0 (choose the AMS package)

   b. Hit *Enter* until you've seen the whole license agreement, then answer *Yes* to accept

   c. Select *Enter* to continue

   d. Choose Option: 1 (yes, add hostname resolution)

      <Budgets server hostname>

      <Budgets server IP address>

   e. Choose Option: 0 (no, stop adding hostname resolution)

   f. Install Location: <Budgets server hostname>

   g. Authentication Server: <authentication daemon hostname>

   h. Authentication Port (this is the port for the authentication daemon, typically `sshd`): <port number>

   i. Provide administrator username: <administrator username>

   j. Install Path: <install path>

6. Once installation completes check the AMS service status:

   `systemctl status altaircontrol`

2.6.4 Enable Passwords in Docker Container Network

To allow administrators and job submitters to use a password to log into Budgets, make it so that Budgets can authenticate users via passwords and `ssh`. This way users can use their password entry in `/etc/passwd` to log into Budgets.

On the service node:

1. Edit `/etc/ssh/sshd_config` and add the following lines:
   ```
   Match Address 10.5.0.0/24
   PasswordAuthentication yes
   ```
2. Make `sshd` reread its configuration file, and restart it:
   ```
   systemctl daemon-reload
   systemctl restart sshd
   ```

2.6.5 Create Certificates for Budgets Daemon Communication Encryption

On the Budgets server host, create the certificates required to encrypt communication between the Budgets and database daemons:

1. Make required directory:
   ```
   cd /home/pbsadmin/
   mkdir budget_certificates
   export AM_DBUSER=<database user; default is pbsdata>
   ```
2. Create a key pair that will serve as both the root CA and the server key pair. This key pair is for 10 years (3650 days):
   ```
   openssl req -new -x509 -days 3650 -nodes -out budget_certificates/ca.crt -keyout
      budget_certificates/ca.key -subj "/CN=root-ca"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/ca.key'
      ...
   ```
3. Create the server key and CSR:
   ```
   openssl req -new -nodes -out server.csr -keyout budget_certificates/server.key -subj "/CN=local-
      host"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/server.key'
      ...
   ```
4. Sign the CSR using the root key:
   ```
   openssl x509 -req -in server.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
      budget_certificates/ca.key -CAcreateserial -out budget_certificates/server.crt
      Signature ok
      subject=/CN=localhost
      Getting CA Private Key
   ```
5. Create client certificate for database user (typically pbsdata):
   ```
   openssl req -new -nodes -out client.csr -keyout budget_certificates/client.key -subj
      "/CN=${AM_DBUSER}"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/client.key'
      ...
   openssl x509 -req -in client.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
      budget_certificates/ca.key -CAcreateserial -out budget_certificates/client.crt
      Signature ok
      subject=/CN=pbsdata
      Getting CA Private Key
   ```
6. Remove unneeded intermediate files:

```
rm -f server.csr client.csr
```

7. Set suitable permissions to protect certificates:

```
chmod og-rwx budget_certificates/*
```

8. View files:

```
cd budget_certificates/
ls -l
    total 28
    -rw-------. 1 root root 1090 Jan 7 14:45 ca.crt
    -rw-------. 1 root root 1704 Jan 7 14:45 ca.key
    -rw-------. 1 root root 17 Jan  7 14:56 ca.srl
    -rw-------. 1 root root 973 Jan  7 14:56 client.crt
    -rw-------. 1 root root 1704 Jan 7 14:55 client.key
    -rw-------. 1 root root 973 Jan  7 14:54 server.crt
    -rw-------. 1 root root 1704 Jan 7 14:52 server.key
pwd
    /home/pbsadmin/budget_certificates
```

2.6.6 Install Budgets Server Module

You can install and configure Budgets using "pbsadmin" as the Budgets administrator username, then switch to a different administrator username later; see section 2.11, “Changing Budgets Administrator to New Username”, on page 58. Install the Budgets server module on the head node (the Budgets server host):

1. Change to new directory created by untarring the main file earlier:

   `cd /var/spool/am/`

2. Run the Budgets installer, and choose the username you want for the Budgets administrator. In our example we use *pbsadmin* for the administrator username:

   `./install -t server -u pbsadmin -c /home/pbsadmin/budget_certificates`

```
    Installing Budgets
    You have selected server.
    AM_EXEC does not exist. Will make it.
    AM_HOME does not exist. Will make it.
    Installing Budgets server...
    **
    **
    Copying source to /opt/am
    **
    **
    Budgets installed successfully.
    sed -i 's/AMADMIN/pbsadmin/g' /opt/am/db/am_database.sql
    sed -i 's/AMADMIN/pbsadmin/g' /opt/am/libexec/am_postinstall
    sed -i 's|@AM_EXEC@|/opt/am|g' /opt/am/libexec/pbs_budget.service
    sed -i 's|@AM_HOME@|/var/spool/am|g' /opt/am/libexec/pbs_budget.service
    sed -i 's|@AM_SERVER@|testbed|g' /opt/am/libexec/pbs_budget.service
    sed -i 's|@AM_PORT@|8000|g' /opt/am/libexec/pbs_budget.service
    cp -rp /opt/am/libexec/pbs_budget.service /usr/lib/systemd/system/pbs_budget.service
```

2.6.7 Set Configuration Parameters

Budgets relies on configuration parameters in the file `/etc/am.conf`. We list the parameters in Table 2-2, “Budgets Configuration Parameters,” on page 44. On the Budgets server host and any client hosts, make sure all of the Budgets configuration parameters are set correctly. Especially make sure that AM_MODE is set to the mode you want, because to change modes you need to restart Budgets. In addition, make sure that AM_AUTH_ENDPOINT and AM_LICENSE_ENDPOINT are set correctly.

2.6.7.1 Budgets Configuration Parameters

Budgets uses the following configuration parameters:

Table 2-2: Budgets Configuration Parameters

Configuration Parameter	Description	Default Value
AM_AUTH_ENDPOINT	Path to AMS module. Format: *<port>@<hostname>*	*9100@<AMS host>*
AM_AUTH_TIMEOUT	Optional. Number of seconds to wait when authenticating. Integer. Set this to a larger value if you have a problem with authentication timeout.	*10*
AM_BALANCE_PRECHECK	Boolean. Directs hook to precheck account balance when job is queued	*False*
AM_DBPORT	Port number for Postgres database	*9876*
AM_DBUSER	Username of database user	*pbsdata*
AM_EXEC	Path to Budgets executables	*/opt/am*
AM_HOME	Home directory for Budgets	*/var/spool/am*
AM_LICENSE_ENDPOINT	Path to license daemon. Format: *<port>@<hostname>*	*6200@<ALM host>*
AM_MODE	Specifies postpaid or prepaid mode	*postpaid*
AM_PORT	Port number on which Budgets listens	*8000*
AM_SERVER	Hostname of Budgets server module host	Host where Budgets server module is installed
AM_WORKERS	Number of workers to spawn. We recommend not changing this setting. For help, contact Altair support.	*2*

2.6.7.2 Example Configuration File

Example of `/etc/am.conf`:

```
AM_PORT=8000
AM_EXEC=/opt/am
AM_HOME=/var/spool/am
AM_WORKERS=2
AM_DBUSER=pbsdata
AM_DBPORT=9876
AM_AUTH_ENDPOINT=9100@my_ams_host
AM_AUTH_TIMEOUT=30
AM_LICENSE_ENDPOINT=6200@my_alm_host
AM_SERVER=my_budget_host
AM_MODE=postpaid
AM_BALANCE_PRECHECK=False
```

2.6.7.3 Caveats and Advice for Budgets Configuration Parameters

- If you change AM_MODE, you must restart Budgets. For procedures to change AM_MODE, see section 3.5, "Changing Between Modes", on page 74.
- Set AM_AUTH_TIMEOUT to a larger value if you have a problem with authentication timeout.

2.6.8 Enable and Start Budgets

- On the Budgets server host, enable and start Budgets:

```
systemctl enable pbs_budget
systemctl start pbs_budget
```

- Check the status of Budgets:

```
systemctl status pbs_budget
```

2.7 Installation Steps for Non-default Location

You must be root to install Budgets. Log in as root.

2.7.1 Set Configuration Parameters

Budgets relies on configuration parameters in the file /etc/am.conf. When you install Budgets in a non-default location, you create the configuration file before you install Budgets. We list the configuration parameters in Table 2-2, "Budgets Configuration Parameters," on page 44. Create the configuration file and make sure that:

- All of the configuration parameters are set correctly
- You set AM_HOME and AM_EXEC to your non-default locations
- You set AM_MODE to the mode you want, because to change modes you need to restart Budgets
- You set AM_AUTH_ENDPOINT and AM_LICENSE_ENDPOINT correctly

Example of /etc/am.conf for non-default locations:

```
AM_PORT=8000
AM_EXEC=/budgets/exec
AM_HOME=/budgets/home
AM_WORKERS=2
AM_DBUSER=pbsdata
AM_DBPORT=9876
AM_AUTH_ENDPOINT=9100@my_ams_host
AM_AUTH_TIMEOUT=30
AM_LICENSE_ENDPOINT=6200@my_alm_host
AM_SERVER=my_budget_host
AM_MODE=postpaid
AM_BALANCE_PRECHECK=False
```

2.7.2 Install Utilities and Docker

1. Install utilities and `docker` on the service node:
   - For CentOS or RedHat:

     Log in as root to the service node (the machine where the AMS module is to be installed).

     ```
     yum install -y yum-utils
     yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
     yum install docker-ce docker-ce-cli containerd.io
     systemctl enable docker
     systemctl start docker
     yum install python3 python3-pip
     yum install openssl
     ```
   - For SLES12 or 15:

     Log in as root to the service node (the machine where the AMS module is to be installed).

     For SLES 12:

     ```
         sudo SUSEConnect -p sle-module-containers/12/x86_64 -r ''
     ```
     For SLES 15:

     ```
         sudo SUSEConnect -p sle-module-containers/15.1/x86_64 -r ''
     sudo zypper install docker
     sudo systemctl enable docker.service
     sudo systemctl start docker.service
     ```
     Configure the firewall to allow forwarding of Docker traffic to the external network:

     ```
     Set FW_ROUTE="yes" in /etc/sysconfig/SuSEfirewall2
     zypper install python3-pip
     ```
   - For Ubuntu:

     Log in as root to the service node (the machine where the AMS module is to be installed).

     ```
     sudo apt-get update
     sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-proper-
         ties-common
     curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
     sudo apt-key fingerprint 0EBFCD88
     ```
     The key should match the second line in the output; validate the last 8 characters. Example of second line:

     9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

     ```
     sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
         $(lsb_release -cs) stable"
     sudo apt-get update
     sudo apt-get install docker-ce docker-ce-cli containerd.io
     sudo apt-get install python3-pip
     ```
     This can install a number of required dependencies, and may take a few minutes.

     ```
     sudo apt-get install openssl
     sudo systemctl enable docker.service
     sudo systemctl start docker.service
     ```

2.7.3 Download Budgets Server and AMS Modules

1. Log in as root to the Budgets server host
2. Go to your selected installation location:

   ```
   # cd <Budgets server installation directory>
   ```

3. On the Budgets server host, download the main file containing both the Budgets server and the AMS module. The filename has the following format:

   PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

   For example:

   PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.gz
4. Untar the main file:

   tar xvfz PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

   For example:

   ```
   tar xvfz PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.gz
   ```

   This creates the following:
   - File named "ams-installer.tar.gz" containing the AMS installer
   - Directory named "am" containing the Budgets server module
5. Copy `ams_installer.tar.gz` over to the service node

2.7.4 Install AMS Module on Service Node

Install the AMS module on the service node. You can start in any directory.

1. Untar the AMS installer file:

   ```
   Tar xvfz ams-installer.tar.gz
   ```

   This gives a directory named "ams-installer", containing:

   - A file named "README.md "
   - A package named "ams-installer.zip"

2. Unzip `ams-installer.zip`:

   ```
   unzip ams-installer.zip
   ```

   This gives a directory named "ams-installation", containing:

   - Directory named "packages" containing the AMS package of folders and configuration files
   - Directory named "pbsworks-packager" containing the Python module that installs AMS

3. Change to `ams-installation` directory:

   ```
   cd ams-installation
   ```

4. Use the Python module to install AMS:

   `python3 -m pip install --upgrade --ignore-installed pbsworks-packager/`
   `/usr/local/bin/pkgr` (Please stay in the AMS installer directory for this step)

5. Answer the questions in the dialogue:

   a. Choose option: 0 (choose the AMS package)
   b. Hit *Enter* until you've seen the whole license agreement, then answer *Yes* to accept
   c. Select *Enter* to continue
   d. Choose Option: 1 (yes, add hostname resolution)
      <Budgets server hostname>
      <Budgets server IP address>
   e. Choose Option: 0 (no, stop adding hostname resolution)
   f. Install Location: <Budgets server hostname>
   g. Authentication Server: <authentication daemon hostname>
   h. Authentication Port (this is the port for the authentication daemon, typically `sshd`): <port number>
   i. Provide administrator username: <administrator username>
   j. Install Path: <install path>

6. Once installation completes check the AMS service status:

   ```
   systemctl status altaircontrol
   ```

2.7.5 Enable Passwords in Docker Container Network

To allow administrators and job submitters to use a password to log into Budgets, make it so that Budgets can authenticate users via passwords and `ssh`. This way users can use their password entry in `/etc/passwd` to log into Budgets.

On the service node:

Edit `/etc/ssh/sshd_config` and add the following lines:

```
Match Address 10.5.0.0/24
PasswordAuthentication yes
```

Make `sshd` reread its configuration file, and restart it:

```
systemctl daemon-reload
systemctl restart sshd
```

2.7.6 Create Certificates for Budgets Daemon Communication Encryption

On the Budgets server host, create the certificates required to encrypt communication between the Budgets and database daemons:

1. Make required directory:
   ```
   cd /home/pbsadmin/
   mkdir budget_certificates
   export AM_DBUSER=pbsdata
   ```
2. Create a key pair that will serve as both the root CA and the server key pair. This key pair is for 10 years (3650 days):
   ```
   openssl req -new -x509 -days 3650 -nodes -out budget_certificates/ca.crt -keyout
      budget_certificates/ca.key -subj "/CN=root-ca"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/ca.key'
      ...
   ```
3. Create the server key and CSR:
   ```
   openssl req -new -nodes -out server.csr -keyout budget_certificates/server.key -subj "/CN=local-
      host"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/server.key'
      ...
   ```
4. Sign the CSR using the root key:
   ```
   openssl x509 -req -in server.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
      budget_certificates/ca.key -CAcreateserial -out budget_certificates/server.crt
      Signature ok
      subject=/CN=localhost
      Getting CA Private Key
   ```
5. Create client certificate for database user (typically pbsdata):
   ```
   openssl req -new -nodes -out client.csr -keyout budget_certificates/client.key -subj
      "/CN=${AM_DBUSER}"
      Generating a 2048 bit RSA private key
      ...
      writing new private key to 'budget_certificates/client.key'
      ...
   openssl x509 -req -in client.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
      budget_certificates/ca.key -CAcreateserial -out budget_certificates/client.crt
      Signature ok
      subject=/CN=pbsdata
      Getting CA Private Key
   ```
6. Remove unneeded intermediate files:

```
rm -f server.csr client.csr
```

7. Set suitable permissions to protect certificates:

```
chmod og-rwx budget_certificates/*
```

8. View files:

```
cd budget_certificates/
ls -l
    total 28
    -rw-------. 1 root root 1090 Jan 7 14:45 ca.crt
    -rw-------. 1 root root 1704 Jan 7 14:45 ca.key
    -rw-------. 1 root root 17 Jan  7 14:56 ca.srl
    -rw-------. 1 root root 973 Jan  7 14:56 client.crt
    -rw-------. 1 root root 1704 Jan 7 14:55 client.key
    -rw-------. 1 root root 973 Jan  7 14:54 server.crt
    -rw-------. 1 root root 1704 Jan 7 14:52 server.key
pwd
    /home/pbsadmin/budget_certificates
```

2.7.7 Install Budgets Server Module

You can install and configure Budgets using "pbsadmin" as the Budgets administrator username, then switch to a different administrator username later; see section 2.11, "Changing Budgets Administrator to New Username", on page 58. Install the Budgets server module on the head node:

1. Change to new directory created by untarring the main file earlier:

   ```
   cd <selected installation directory>/am/
   ```

2. Run the Budgets installer, and choose the username you want for the Budgets administrator. In our example we use *pbsadmin* for the administrator username:

   ```
   ./install -t server -u pbsadmin -c /home/pbsadmin/budget_certificates
   ```

   For example, if you use the example configuration file in section 2.7.1, "Set Configuration Parameters", on page 45, you will see the following:

   ```
       Installing Budgets
       You have selected server.
       AM_EXEC does not exist. Will make it.
       AM_HOME does not exist. Will make it.
       Installing Budgets server...
       **
       **
       Copying source to /budgets/exec
       **
       **
       Budgets installed successfully.
       sed -i 's/AMADMIN/pbsadmin/g' /budgets/exec/db/am_database.sql
       sed -i 's/AMADMIN/pbsadmin/g' /budgets/exec/libexec/am_postinstall
       sed -i 's|@AM_EXEC@|/budgets/exec|g' /budgets/exec/libexec/pbs_budget.service
       sed -i 's|@AM_HOME@|/budgets/home|g' /budgets/exec/libexec/pbs_budget.service
       sed -i 's|@AM_SERVER@|testbed|g' /budgets/exec/libexec/pbs_budget.service
       sed -i 's|@AM_PORT@|8000|g' /budgets/exec/libexec/pbs_budget.service
       cp -rp /budgets/exec/libexec/pbs_budget.service /usr/lib/systemd/system/pbs_budget.service
   ```

2.7.8 Enable and Start Budgets

- On the Budgets server host, enable and start Budgets:

  ```
  systemctl enable pbs_budget
  systemctl start pbs_budget
  ```

- Check the status of Budgets:

  ```
  systemctl status pbs_budget
  ```

2.8 Validating Budgets

1. Log in as pbsadmin
2. Test authentication:

   ```
   amgr login
   ```
3. List users (pbsadmin and amteller):

   ```
   amgr ls user -l
   ```

2.9 Configuring Budgets for Failover

Budgets uses Pacemaker and Corosync to manage failover for the Budgets and data service daemons. You set up a primary Budgets server and a secondary Budgets server, and a shared filesystem used by either one and available to both, as a *failover cluster*. The primary normally handles all server traffic, but the secondary takes over and becomes active if the primary becomes unavailable. Failover for the Budgets daemon and the data service daemon happens together. Budgets starts the data service on the same host where the Budgets daemon runs. Pacemaker and Corosync manage the flow of traffic so that it goes to the active server and database.

2.9.1 Prerequisites for Configuring Budgets Failover

2.9.1.1 Third-party Software Prerequisites

- Pacemaker 1.1.23
- Corosync 2.4.5 or later
- pcs 0.9.169

2.9.1.2 Budgets Server Host Prerequisites

- Budgets server hosts must be identical
- Both server hosts must have access to shared filesystem
- Each server host should have two network connections:
  - Dedicated private ethernet connection that Pacemaker and Corosync use to manage host state
  - Public network that Budgets client commands will use for communication traffic with Budgets server
- Each server host should have two hostnames:
  - Private hostname for use with private ethernet connection, for example "PrimaryFailover" and "SecondaryFailover"
  - Public hostname for use with client commands, for example "Primary" and "Secondary"
- Same version of Budgets on both servers
- Accounts for administrator, investors, managers, and teller must be identical on both hosts

2.9.1.3 Filesystem Prerequisites

- Shared `AM_HOME` filesystem on a third host, that is mounted on both Budgets server hosts and always available to both hosts
- No root squash on shared filesystem
- Hostname resolution must work both ways between Budgets server hosts, and between both servers and any other hosts involved, for example PBS server hosts
- The `AM_HOME` directory must be readable and writable from both servers by the Budgets administrator

2.9.1.4 Optional

- STONITH script, if required. The administrator writes this script.

2.9.1.5 Notes

- There are no prerequisites for the service node.

2.9.2 Installing Corosync, Pacemaker, and pcs

On each Budgets server host:

1. Log into each Budgets server host as root.
2. Install Budgets if it is not already installed. See section 2.6, "Installation Steps for Default Location", on page 37.
3. Install Pacemaker, Corosync, and pcs.
4. When you install Pacemaker, the installation creates the hacluster account, which is the proxy user for Pacemaker. You need to give hacluster a password on each server host:

   ```
   passwd hacluster
   <password for hacluster account>
   ```
5. Create a `budget` directory in the `ocf` library:

   ```
   mkdir /usr/lib/ocf/resource.d/budgetmanager
   ```
6. Copy `ocf` files `am_ocf` and `am_data_ocf` from the Budgets package to the `budget` directory:

   ```
   cp /root/am/failover/budgetmanager/* /usr/lib/ocf/resource.d/budgetmanager/
   ```
7. Set the permissions for the `budget` directory:

   ```
   chmod -R +x /usr/lib/ocf/resource.d/budgetmanager
   ```

2.9.3 Configuring Pacemaker

1. If Budgets is running on any host, stop Budgets:
   ```
   systemctl disable pbs_budget
   systemctl stop pbs_budget
   ```
2. Log into the primary Budgets server host as root
3. Authorize Pacemaker to use the server hosts, using their private hostnames:
   ```
   pcs cluster auth <private primary hostname> <private secondary hostname>
   ```
4. Create the failover cluster, name it "am_cluster", and start it, using the private hostnames:
   ```
   pcs cluster setup --start --name am_cluster <private primary hostname> <private secondary host-
       name>
   ```
5. Set the quorum policy to *ignore*:
   ```
   pcs property set no-quorum-policy=ignore
   ```
6. Set whether or not stonith is enabled. If you have a STONITH script and want to use it, you can enable stonith. If not, disable it:
   ```
   pcs property set stonith-enabled=false
   ```
7. Create the virtual_ip resource to represent a virtual IP address. This IP address should be exclusively allocated for this purpose and not used by any physical host on your network. This is the IP address that client commands will use to connect to the active Budgets server over the public network. Pacemaker and Corosync direct traffic from this IP address to whichever server is active:
   ```
   pcs resource create virtual_ip ocf:heartbeat:IPaddr2 ip=<virtual IP address> cidr_netmask=32
       nic=<nic-name-of-public-network> op monitor interval=10s
   ```
8. Create the amserver resource to represent the Budgets daemon. Note that here we are assuming `am.conf` is in `/etc`. Make sure you use the appropriate location:
   ```
   pcs resource create amserver ocf:budgetmanager:am_ocf conf=/etc/am.conf op monitor interval=15s
   ```
9. Add a constraint to Pacemaker so that it always keeps the virtual_ip and amserver resources on the same host. When one moves, the other goes with it:
   ```
   pcs constraint colocation add amserver virtual_ip INFINITY
   ```
10. Make the primary server host be the preferred server host:
    ```
    pcs constraint location amserver prefers <primary>
    ```
11. Budgets is automatically started on the primary server host. You do not have to start Budgets.

2.9.4 Caveats and Recommendations for Failover

- Make sure only one Budgets server daemon is running at a time. Do not start a second Budgets server. If two instances of Budgets are active at the same time, the database will become corrupted.
- Make sure that the shared `AM_HOME` directory is always available on both the primary and secondary server hosts. Do not prevent either host from reaching `AM_HOME`.

2.9.5 Starting, Stopping, and Getting Status of Budgets with Failover Configured

- To start Budgets when failover is configured:

  On each Budgets server host, in any order:

  ```
  pcs cluster start
  ```
- To stop Budgets when failover is configured, first stop the secondary, then the primary:

  ```
  pcs cluster stop <secondary server host>
  pcs cluster stop <primary server host>
  ```
- To check the status of Budgets when failover is configured:

  ```
  pcs cluster status
  ```

2.10 Upgrading Budgets

1. Log in to the Budgets server as root
2. Turn off scheduling in all associated PBS Professional complexes
3. Disable all PBS queues:

   ```
   qmgr -c 'set queue <queue name> enabled=false'
   ```
4. Allow all jobs to finish, or kill them
5. If necessary, reconcile all jobs:

   ```
   amgr reconcile [options]
   ```

   See section 4.3.6, “Reconciling Service Units”, on page 130
6. Stop Budgets (ideal for backup, but not essential)

   ```
   systemctl stop pbs_budget
   ```
7. Back up `/var/spool/am`
8. Make a copy of the formula file used for each cluster. If you don't have these on hand, export them at each PBS server host:

   ```
   qmgr -c 'export hook am_hook application/x-config default' > am_hook.json
   ```
9. If necessary, create certificates; see Create Certificates for Budgets Daemon Communication Encryption
10. Allow Budgets to work with PBS Professional by updating `sudoers` file; see section 2.5.2, “Allow Interaction with PBS Professional”, on page 33
11. Untar the new install package; this creates the `am` directory
12. Change to the `am` directory and install Budgets (we include the **`-c /home/pbsadmin/budget_certificates/`** option here so that the installer uses the new certificates; if you have not changed the certificates you can leave this option out):

    ```
    cd am
    ./install -t upgrade -c /home/pbsadmin/budget_certificates/
    ```
13. Installer asks whether scheduling is disabled; enter "*Y*"

    Installer actions:

    - The installer backs up your database. If this fails, it aborts the upgrade. If the upgrade fails, the installer restores the backup you just made, and you can reinstall the older version of Budgets to return to operation. You can install it in place over the existing data, and it will pick up where it left off.
    - After backing up the data, the installer uninstalls the old Budgets, installs the new one, updates the database schema and updates the data. This could take some minutes if there is a lot of data.
    - The installer updates `am.conf`.
14. Make sure that all of the settings in section 2.6.7.1, “Budgets Configuration Parameters”, on page 44 are set correctly. Note that after an upgrade, the default for AM_MODE is *prepaid*.
15. For each PBS complex, create and configure the am_finished_job resource:

    ```
    qmgr -c "c r am_finished_job type=string"
    qmgr -c "set server resources_available.am_finished_job=NA"
    ```
16. Restart Budgets:

    ```
    systemctl restart pbs_budget
    ```

17. This version of Budgets comes with a new hook in a .py file. Configure the hooks; see section 3.2.4, “Create and Configure Budgets Hooks”, on page 69
18. Update the formulas for each cluster with the file from its PBS complex:

    ```
    amgr update cluster -n <PBS server> -f <formula filename>
    ```

19. Enable all PBS queues:

    ```
    qmgr -c 'set queue <queue name> enabled=true'
    ```

20. Enable scheduling at each associated PBS complex.

Budgets is ready to track and manage service units.

2.11 Changing Budgets Administrator to New Username

You can install and configure Budgets using one administrator username, then switch to another later. To switch to a new administrator username:

1. Make sure the new administrator account meets the criteria in section 2.5.1.1, “Budgets Administrator”, on page 32.

- Create the new administrator username, and set a password:

  ```
  adduser -u 901 <new administrator username>
  passwd <new administrator username> <password>
  ```

2. Add the new account to Budgets, and assign it the *admin* role:

   ```
   amgr add user -n <new administrator username> -r admin -A  <accounting policy> -c <PBS server>  -r
       <role> [-h <group list>] [ -a <active>]
   ```

3. Permanently disable pbsadmin.

   Optional. This is **not recoverable**. You can remove *admin* role from pbsadmin, or deactivate pbsadmin:

   ```
   amgr update user -n pbsadmin -r <new role>
   amgr update user -n pbsadmin -a False
   ```

2.12 Installing Budgets Client Module

You can install just the Budgets client commands on other machines besides the Budgets server host.

2.12.1 Prerequisites for Installing Budgets Client Commands

Any host where you install just the client commands must be able to reach the Budgets server, and must be a supported platform for the Budgets client commands (see the *PBS Professional Release Notes*).

2.12.2 Caveats and Restrictions for Installing Budgets Client Commands

When you extract the client command package, you create a directory with the same name as the one created when you extract the server package. This can overwrite your server directory.

2.12.3 Steps to Install Budgets Client Commands

On any host where you will run only the Budgets client commands:

1. Log in as root
2. Extract the client package:

   # tar -xzvf PBSPro-budget-client_<release number>-<OS>_<chipset>.tar.gz

   For example:

   ```
   # tar -xzvf PBSPro-budget-client_2022.1.0-CentOS7_x86_64.tar.gz
   ```
3. Change to the directory you just created:

   ```
   # cd am
   ```
4. Install the client commands:

   ```
   # ./install -t client
   ```
5. Make sure that `/etc/am.conf` is identical to the one on the Budgets server host. For example:

   ```
   # cat /etc/am.conf
   AM_PORT=8000
   AM_EXEC=/opt/am
   AM_HOME=/var/spool/am
   AM_WORKERS=2
   AM_DBUSER=pbsdata
   AM_DBPORT=9876
   AM_AUTH_ENDPOINT=9100@my_ams_host
   AM_AUTH_TIMEOUT=30
   AM_LICENSE_ENDPOINT=6200@my_alm_host
   AM_SERVER=my_budget_host
   AM_MODE=postpaid
   AM_BALANCE_PRECHECK=False
   ```
6. Test the client:

   ```
   # su - pbsadmin
   $ amgr login
   Password: ******
   $ amgr ls user -l
   ```

3 Configuring and Managing Budgets

3.1 Defining Billing Periods

Choose and define your billing periods. We describe billing periods in section 1.7.1, "Periods, Allocation Periods, Billing Periods", on page 18.

Make sure that periods at the same level do not overlap. For example, if Quarter1 ends March 31st, make sure that Quarter2 does not begin sooner than April 1st.

If you want to create periods with a parent-child relationship, you must create the parent period first. You cannot add a parent to an existing child. For example, if you want Year as the parent and Quarter1, Quarter2, etc., as children, create Year first.

If you create child periods, make sure that they fit within the parent period.

Make sure that your period hierarchy is finalized BEFORE doing any transactions or running any jobs; you cannot update or remove periods once transactions have been performed or jobs have started.

To define each billing period, add it to Budgets:

amgr add period -n <period name> -S <start date> -E <end date> [-p <name of parent period>]

See section 4.2.1.6, "Adding a Period", on page 83.

3.2 Adding a PBS Complex and Setting its Billing Model

Budgets interacts with the PBS server via two hooks and hook configuration file containing the billing formulas. Budgets uses two identical hooks, named am_hook and am_hook_periodic; both hooks use the same configuration file. The formulas define the *billing model* for that PBS complex. You can set different billing formulas for each PBS complex.

To use a PBS complex with Budgets:

- At the PBS server host, the administrator creates and configures the hooks, and defines the formulas to use when billing
- At the Budgets server host, the administrator adds a cluster data structure to Budgets that will represent the PBS complex, and sets the formulas at the cluster to be the same as the one at the PBS complex

3.2.1 Caveats and Advice on Billing Model

For jobs that run on multiple vnodes, the hook uses the sum of the resources used. The hook does not support calculation of variable rates on multiple vnodes.

In postpaid mode, all transactions are allocated to top-level periods.

3.2.2 Steps to Add Complex and Set Billing Model

1. Log into the Budgets server host
2. Optionally modify the formula file as desired; see section 3.2.3, “Define Billing Formulas”, on page 62
3. Log into the PBS server host
4. If the PBS server host is different from the Budgets server host, copy the hook and formula files from your Budgets installation to the PBS server host
5. At the PBS server host, create and configure the Budgets hooks; see section 3.2.4, “Create and Configure Budgets Hooks”, on page 69
6. At the Budgets server host, add a cluster data structure that will represent the PBS complex and specify the formulas you used for the PBS complex:

   amgr add cluster -n <PBS server> -f <formula filename>

   For example, to add a PBS complex whose server is named HPC1, and use the formulas defined in `formula_hpc1.json`:

   ```
   amgr add cluster -n HPC1 -f formula_hpc1.json
   ```

   See section 4.2.1.5, “Adding a Cluster”, on page 83.
7. At the PBS server host, specify whether you want to separate on premise and cloud costs. See section 3.2.6, “Separating On Premise and Cloud Costs”, on page 71.

3.2.3 Define Billing Formulas

Budgets uses billing formulas, in which you define how service units are calculated. These formulas can be different for each cluster.

3.2.3.1 Billing Formula File and Format

The billing model is defined in a billing formula file in JSON format. We provide a default billing formula file named am_hook.json; you can modify it to fit your needs. Name it <formula file>.json.

You can create formulas using Python.

You must include this line:

```
"auth_user": "amteller",
```

3.2.3.1.i Constants (Numbers)

To use a constant (a number) in a formula, you have to define a constant for it in the "constants" section. You cannot use numbers directly in a formula.

Constants must start with the prefix "CONST_". Make sure you define each constant as a floating point number.

Example 3-1: Formula with variables and a constant

```
{
    "auth_user": "amteller",
    "constants": {
        "cpu_count": "job.ncpus",
        "time_span": "job.walltime",
        "CONST_time": 0.01667
    },
    "formulas": {
        "cpu_hrs": "cpu_count*time_span*CONST_time"
    }
}
```

3.2.3.1.ii PBS Resources and Attributes

If you want to use a PBS resource or attribute as a variable in a formula, you have to define a constant to represent it in the "constants" section. For example, instead of using the ncpus resource directly, define a constant named "cpu_count", and set it to the value of the ncpus resource, as we have done in "Formula with variables and a constant".

You cannot use a resource or attribute directly in a formula. If the value of a PBS attribute or resource has not been set, you cannot access it in a formula.

You can use the following PBS Professional elements:

- Job, server, queue, and node attributes that are of type float or integer
- Built-in or custom resources that are of type float or integer
- Built-in resources that are of type duration: walltime, cput, and eligible_time
- Built-in resources that are of type size: mem and vmem

Note that if the value of an attribute or resource has not been set, you cannot access it.

The syntax for specifying resources in the formula file is different from the syntax you would use in a normal hook:

- Before and while running, job resources are read from the job's Resource_List attribute. After the job runs, job resources are read from the job's resources_used attribute. The syntax for specifying a job resource in the formula file is different from other hook operations:

    job.<resource name>

    For example, to specify what would be job.Resource_List["ncpus"]:

    ```
    job.ncpus
    ```
- Server, queue, and vnode resources are read from the resources_available attribute. The syntax for specifying resources at the server, queue, or vnode is slightly different from other hook operations (you omit the quotes):

    <object>.resources_available[<resource name>]

    For example, to use the value of what would be server.resources_available["charge_rate"] in the usual hook syntax:

    ```
    server.resources_available[charge_rate]
    ```

For jobs that run on multiple vnodes, the hook uses the sum of the resources used across all sister vnodes.

Example 3-2: Formula using job resources and a queue attribute:

```
{
    "auth_user": "amteller",
    "constants": {
        "cpu_count": "job.ncpus",
        "gpu_count": "job.ngpus",
        "time_span": "job.walltime",
        "queue_priority": "queue.Priority",
        "CONST_threshold_prio": 1,
        "CONST_low": 0.5,
        "CONST_high": 1.0
    },
    "formulas": {
        "cpu_hrs": "((cpu_count+gpu_count)*time_span) *
            (queue_priority < CONST_threshold_prio and CONST_low or CONST_high)"
    }
}
```

3.2.3.1.iii Operators

- Logical operators (AND and OR)
- Conditional operators (>,>=,<,<=,==,!=).

3.2.3.1.iv Units

The PBS resources walltime, cput, and eligible_time are in *duration* format (hh:mm:ss). The hook is give the value of these in seconds when used in the formula for the service unit. For example, if a job's walltime is two minutes, the value used in the formula is 120.

The PBS resources mem and vmem are in *size* format (3b, 20kb etc.). The hook converts their values to kb in the service unit formula.

The following table lists the units you can use in a billing formula:

Table 3-1: Units in Billing Formulas

Formula Item	Units	Example
Constants	*Floating point*	12.34
Time-based PBS resources, e.g. walltime, cput, eligible_time	*Integer seconds*	120 (2 minutes)
Memory PBS resources, e.g. mem, vmem	*kb*	1048576kb (1GB)
ncpus, represented as cpu_count	*Integer*	4 (ncpus=4)

When Budgets computes cost data for a cloud job, it uses the data you gave to Budgets via `amgr update cloud-data`. This data includes cost per unit time, and specifies the time unit for the updated data. However, the units are not automatically translated when computing formula costs, so you need to make sure that you handle the time units correctly. For example, if you reported cost data in minutes but your formula produces CPU hours, you need to convert the time elements explicitly. To find out what units are used for the cloud cost data, you can query Budgets via `amgr ls clouddata`; see section 4.2.2.10, “Listing Cloud Data”, on page 90.

3.2.3.1.v Distinguishing Cloud Costs from On Premise Costs

Budgets has the following reserved words that you can use in formulas to indicate how cloud or on premise job costs should be calculated:

IS_CLOUD_JOB

Flag to indicate whether or not this calculation applies to a cloud job. Set internally by Budgets. Used by Budgets when applying this formula to a job. When job's am_job_cache resource contains *False*, Budgets sets this to *0*.

Values:

0: Job is not a cloud job

1: Job is a cloud job

Default: *0*

IS_ONPREMISE_JOB

Flag to indicate whether or not this calculation applies to an on premise job. Set internally by Budgets. Used by Budgets when applying this formula to a job. When job's am_job_cache resource contains *False*, Budgets sets this to *1*.

Values:

0: Job is not an on premise job

1: Job is an on premise job

Default: *1*

CLOUD_COST

Cloud cost per reported CPU. Set internally by Budgets, using cost information you provide to Budgets via `amgr update clouddata` (see section 4.2.3.10, “Updating Cloud Cost Data”, on page 98).

Default: *1*

3.2.3.2 Defining Service Units

In order to use a standard service unit, you must define it in the formula file. You define the formula for each service unit in the "formulas" section. Make sure that the name you use for the service unit in a formula is identical to the name you use in the `amgr add serviceunit` command. The service unit name should consist only of alphanumeric and underscore characters; blank spaces are not allowed.

You do not need to define dynamic service units in the formula file.

See section 1.7.2, “Service Units”, on page 18.

3.2.3.3 Defining Cloud and On Premise Service Units

You can calculate separate costs for cloud and on premise jobs by using the reserved words IS_CLOUD_JOB, IS_ONPREMISE_JOB, and CLOUD_COST. You can create formulas that handle both cloud and on premise jobs.

Use IS_CLOUD_JOB and IS_ONPREMISE_JOB as on-off switches to control whether a cost contributes to what is computed. When calculating cloud costs, include IS_CLOUD_JOB in each formula for cloud costs. When calculating on premise costs, include IS_ONPREMISE_JOB in each formula for on premise costs.

Budgets examines the value of the job's am_job_cache resource to see how the server resource am_cloud_enabled was set when the job was submitted. The value of the server resource is recorded in the job resource at the time of job submission.

When the recorded value for am_cloud_enabled is *True*, and this is a cloud job, Budgets treats this job as a cloud job, and sets IS_ONPREMISE_JOB to *0*, and sets IS_CLOUD_JOB to *1*, for this job.

When the recorded value for am_cloud_enabled is *False*, Budgets treats this job as on premise (whether or not it is a cloud job), and sets IS_ONPREMISE_JOB to *1*, and sets IS_CLOUD_JOB to *0*, for this job.

For cloud jobs, use CLOUD_COST as the cost per unit time for an instance. Budgets calculates the value of CLOUD_COST using the cloud cost data you provided via `amgr update clouddata`. When calculating the cost for a cloud job, Budgets uses the cost for the instance that the job would use.

3.2.3.3.i Example of Defining Cloud and On Premise Service Units

Example 3-3: Define the following service units:

- onpremise_cpu_hrs: number of CPU hours on premise
- cloud_cpu_hrs: number of CPU hours in the cloud
- onpremise_dollar: on premise CPU hours, multiplied by cost per unit time for those CPUs
- cloud_dollar: cloud CPU hours, multiplied by cost per unit time for those CPUs
- dollar: total cost for on premise and cloud jobs; on premise dollars plus cloud dollars

We also define CONST_onpremise_cpu_cost as the cost per unit time for on premise CPUs.

Formula file:

```
{
    "auth_user": "amteller",
    "constants":{
        "ncpus" : "job.ncpus",
        "walltime" : "job.walltime",
        "CONST_onpremise_cpu_cost" : 0.05
    },
    "formulas":{
        "onpremise_cpu_hrs" : "(ncpus*walltime)*IS_ONPREMISE_JOB",
        "cloud_cpu_hrs" : "(ncpus*walltime)*IS_CLOUD_JOB",
        "onpremise_dollar" : "(ncpus*walltime*CONST_onpremise_cpu_cost)*IS_ONPREMISE_JOB",
        "cloud_dollar" : "(ncpus*walltime*CLOUD_COST)*IS_CLOUD_JOB",
        "dollar" : "(ncpus*walltime*CONST_onpremise_cpu_cost*IS_ONPREMISE_JOB)
                     +(ncpus*walltime*CLOUD_COST*IS_CLOUD_JOB)"
    }
}
```

If we use the formulas in this example, and have a job with:

- ncpus = 2
- walltime = 10

Where

- CONST_onpremise_cost = 0.05
- CLOUD_COST = 0.1

Then, for an on premise job:

- IS_CLOUD_JOB = 0
- IS_ONPREMISE_JOB = 1
- onpremise_cpu_hrs = (2*10)*1 = 20
- cloud_cpu_hrs = (2*10)*0 = 0
- onpremise_dollar = (2*10*0.05)*1 = 1
- cloud_dollar = (2*10*0.1)*0 = 0
- dollar = (2*10*0.05)*1 + (2*10*0.1)*0 = 1+0 = 1

And for a cloud job:

- IS_CLOUD_JOB = 1
- IS_ONPREMISE_JOB = 0
- onpremise_cpu_hrs = (2*10)*0 = 0
- cloud_cpu_hrs = (2*10)*1 = 20
- onpremise_dollar = (2*10*0.05)*0 = 0
- cloud_dollar = (2*10*0.1)*1 = 2
- dollar = (2*10*0.05)*0 + (2*10*0.1)*1 = 0+2 = 2

3.2.3.4 Default Billing Formula File Contents

This is the contents of the default am_hook.json configuration file used by the Budgets hook. The default service unit is cpu_hrs.

```
{
    "auth_user" : "amteller",
    "constants":{
        "cpu_count": "job.ncpus",
        "time_span": "job.walltime"
    },
    "formulas":{
        "cpu_hrs": "cpu_count*time_span"
    }
}
```

3.2.3.5 Formula File Examples

Example 3-4: Configuration file using multiple constants and operators. We define the constants CONST_a, CONST_b, and CONST_prio. We use them in the formulas section with logical operators 'and', 'or', the comparison operator '>', and the arithmetic operator '*'.

```
{
    "auth_user" : "amteller",
    "constants":{
        "cpu_count": "job.ncpus",
        "time_span": "job.walltime",
        "gpu_count": "job.ngpus",
        "queue_priority": "queue.Priority",
        "node_cpus": "node.resources_available[ncpus]",
        "CONST_a": 2.0,
        "CONST_b": 1.0,
        "CONST_prio": 150
    },
    "formulas":{
        "cost":"(gpu_count and node_cpus or cpu_count)
            *time_span*(queue_priority > CONST_prio and CONST_a or CONST_b)"
    }
}
```

Example 3-5: Formula with three different service units called cpu_hrs, gpu_hrs, and mics_hrs:

```
{
    "auth_user" : "amteller",
    "constants":{
        "cpu_count" : "job.ncpus",
        "time_span" : "job.walltime",
        "gpu_count" : "job.ngpus",
        "mic_count" : "job.nmics"
    },
    "formulas":{
        "cpu_hrs" : "cpu_count*time_span",
        "gpu_hrs" : "gpu_count*time_span",
        "mics_hrs" : "mic_count*time_span"
    }
}
```

3.2.4 Create and Configure Budgets Hooks

1. Log into the PBS server host.
2. If the PBS server host is different from the Budgets server host, copy the hook and configuration files to the PBS server host.
3. Create and configure the am_hook and am_hook_periodic hooks:
   - For prepaid mode:
     ```
     qmgr -c "c h am_hook"
     qmgr -c "s h am_hook order=1000"
     qmgr -c "i h am_hook application/x-python default  /opt/am/hooks/pbs/am_hook.py"
     qmgr -c "i h am_hook application/x-config default /opt/am/hooks/pbs/<formula file>.json"
     qmgr -c "s h am_hook enabled=true"
     qmgr -c "s h am_hook alarm=90"
     qmgr -c "c h am_hook_periodic"
     qmgr -c "s h am_hook_periodic event=periodic"
     qmgr -c "s h am_hook_periodic freq=120"
     qmgr -c "i h am_hook_periodic application/x-python default  /opt/am/hooks/pbs/am_hook.py"
     qmgr -c "i h am_hook_periodic application/x-config default /opt/am/hooks/pbs/<formula
         file>.json"
     qmgr -c "s h am_hook_periodic enabled=true"
     ```
   - For prepaid mode when am_cloud_enabled = *False*:
     ```
     qmgr -c "s h am_hook event='queuejob,runjob,modifyjob,movejob'"
     ```
   - For prepaid mode when am_cloud_enabled = *True*:
     ```
     qmgr -c "s h am_hook event='queuejob,runjob,modifyjob,movejob,execjob_epilogue'"
     ```
   - For postpaid mode:
     ```
     qmgr -c "c h am_hook"
     qmgr -c "s h am_hook event='queuejob,modifyjob,movejob'"
     qmgr -c "s h am_hook order=1000"
     qmgr -c "i h am_hook application/x-python default  /opt/am/hooks/pbs/am_hook.py"
     qmgr -c "i h am_hook application/x-config default /opt/am/hooks/pbs/<formula file>.json"
     qmgr -c "s h am_hook enabled=true"
     qmgr -c "s h am_hook alarm=90"
     qmgr -c "c h am_hook_periodic"
     qmgr -c "s h am_hook_periodic event=periodic"
     qmgr -c "s h am_hook_periodic freq=120"
     qmgr -c "i h am_hook_periodic application/x-python default  /opt/am/hooks/pbs/am_hook.py"
     qmgr -c "i h am_hook_periodic application/x-config default /opt/am/hooks/pbs/<formula
         file>.json"
     qmgr -c "s h am_hook_periodic enabled=true"
     ```

3.2.4.1 Caveats and Advice on Budgets Hooks

- Do not try to combine the Budgets hooks into one hook.
- The am_hook has to run after all the other hooks for each event. Make sure it has the highest order. For example, you can set it to a high order value such as 1000:

  ```
  qmgr -c "s h am_hook order=1000"
  ```
- The difference between postpaid and prepaid modes is that am_hook has a runjob event in prepaid mode but not in postpaid mode.

3.2.5 Configuring Resources for Budgets

3.2.5.1 List of PBS Professional Custom Resources for Budgets

Budgets uses the following custom resources:

am_cloud_enabled

Boolean. Set at server. Value of this resource is recorded in each job's am_job_cache resource at the time of job submission.

When you set this to *True*, add the execjob_epilogue event to the am_hook list of trigger events.

Set by administrator.

am_job_amount

String. Set by Budgets. Used for reporting costs.

am_job_cache

String in JSON format. This resource is included in each job's resource request. Value of this string includes the value for the server am_cloud_enabled resource at the time of job submission. Set and used by Budgets.

am_job_quote

Boolean. Set by job submitter. Indicates that Budgets should calculate cost estimate for a job and return that estimate to the job submitter. When this boolean is included in a job submission command, the job is not submitted.

am_finished_job

String. Set by Budgets.

am_node_cache

String. Set by Budgets.

3.2.5.2 Create and Set Resources

```
qmgr -c "c r am_cloud_enabled type=boolean"
qmgr -c "c r am_job_amount type=string"
qmgr -c "c r am_job_cache type=string,flag=m"
qmgr -c "c r am_job_quote type=boolean"
qmgr -c "c r am_finished_job type=string"
qmgr -c "c r am_node_cache type=string"
qmgr -c "set server resources_available.am_finished_job=NA"
```

3.2.6 Separating On Premise and Cloud Costs

You can use separate cost data and formulas for on premise and cloud jobs, or you can treat all jobs as if they are on premise jobs. This behavior is controlled by the value of the am_cloud_enabled server resource at the time each job is submitted. Budgets uses this resource as follows:

- When a job is submitted, the value of the am_cloud_enabled server resource is included in the data captured in the am_job_cache resource for that job
- When Budgets operates on a job, it uses the value of am_cloud_enabled stored in the job's am_job_cache resource to decide how to treat the job.
  - If the recorded value is *True*, and the job is a cloud job, costs for the job can be computed using cloud cost data, and formulas for cloud jobs are applied to this job
  - If the recorded value is *False*, the job is treated like an on premise job regardless of whether or not it is a cloud job. Costs for the job can be computed using on premise cost data, and formulas for on premise jobs are applied to this job

3.2.6.1 Behavior When Separating Costs in Prepaid Mode

Budgets examines the value of each job's am_job_cache resource, and acts accordingly.

When Budgets is in prepaid mode and a job's am_job_cache contains *True*:

- If the job is a cloud job, you can employ formulas specifically for cloud costs
  - Budgets automatically sets IS_CLOUD_JOB to *1* for this job
  - Budgets can use cloud cost data in formulas
- If the job is a cloud job, the job submitter can get a quote for this job using cloud cost data
- The job can run only when it has sufficient credit

When Budgets is in prepaid mode a job's am_job_cache contains *False*:

- Budgets behaves as if this is not a cloud job; this job is treated like an on premise job in formulas
  - Budgets sets IS_CLOUD_JOB to *0* for this job
  - Budgets sets IS_ONPREMISE_JOB to *1* for this job
- The job submitter can get a quote for this job using on premise data only
- The job can run only when it has sufficient credit

3.2.6.2 Behavior When Separating Costs in Postpaid Mode

When Budgets is in postpaid mode and a job's am_job_cache is *True*:

- You can employ formulas specifically for cloud costs
  - If this is a cloud job, Budgets automatically sets IS_CLOUD_JOB to *1* for this job
- The job submitter can get a quote for this job using cloud cost data
- The job can run regardless of credit

When Budgets is in postpaid mode and a job's am_job_cache is *False*:

- Budgets behaves as if this is not a cloud job; this job is treated like an on premise job in formulas
  - Budgets sets IS_CLOUD_JOB to *0* for this job
  - Budgets sets IS_ONPREMISE_JOB to *1* for this job
- The job submitter can get a quote for this job using on premise data only
- The job can run regardless of credit

3.2.6.3 Steps to Separate On Premise and Cloud Costs

To separate on premise and cloud costs:

- Set the am_cloud_enabled server-level Boolean resource to *True*:

```
qmgr -c "set server resources_available.am_cloud_enabled=true"
```

- Add the execjob_epilogue trigger event to am_hook:

```
qmgr -c "s h am_hook event='queuejob,runjob,modifyjob,movejob,execjob_epilogue'"
```

3.2.7 Requiring Sufficient Credit Before Bursting Cloud Nodes

You can require that job owners have sufficient credit before allowing cloud nodes to be burst for jobs, by setting the server's am_cloud_enabled resource to *True*. PBS Cloud examines the value of the server's am_cloud_enabled resource, and acts accordingly.

3.2.7.1 Behavior When Requiring Credit in Prepaid Mode

When Budgets is in prepaid mode and the server's am_cloud_enabled is *True* and a job's am_job_cache contains *True*, if this is a cloud job, PBS Cloud will burst nodes for this job only if it has sufficient credit, and that credit is calculated using cloud costs.

When Budgets is in prepaid mode and the server's am_cloud_enabled is *True* and a job's am_job_cache contains *False*, if this is a cloud job, PBS Cloud will burst nodes for this job only if it has sufficient credit, but that credit is calculated using on premise costs.

3.2.7.2 Behavior When Requiring Credit in Postpaid Mode

When Budgets is in postpaid mode, the value of the server's am_cloud_enabled is *True* does not affect the behavior of PBS Cloud. If this is a cloud job, PBS Cloud will burst nodes for this job when it is ready to run, regardless of credit.

3.2.8 Allow Easy Quote Request

We recommend making it easy for job submitters to get job quotes. Provide a wrapper for the quote request, and make it accessible to all job submitters. Add the following to /etc/bashrc (or the equivalent). The trailing space is important:

```
alias quote='qsub -l am_job_quote=true '
```

3.2.9 Changing the Billing Formula File

3.2.9.1 Procedure for Changing Billing Formula File

To change a formula you use at a PBS complex:

1. Drain all jobs from the PBS complex
2. Modify the formula
3. Update the complex and its cluster with the new formula file:

amgr update cluster -n <PBS server> -f <formula filename>

For example, to change the formula file for a PBS complex whose server is named HPC1, as well as its associated cluster, so that they use the formula defined in `new_formula_hpc1.json`:

```
amgr update cluster -n HPC1 -f new_formula_hpc1.json
```

See section 4.2.3.5, “Updating Clusters”, on page 96

3.2.9.2 Caveats and Restrictions on Changing Billing Formula File

- Do not add resources to a formula while jobs are running. If you add new resources to a formula used by a PBS complex while jobs are running, any running jobs will fail to reconcile because they won't include the required data in their cache to allow Budgets to bill for the new resources.
- Do not change a formula while jobs are running. If you change a formula while jobs are running, users with running jobs may be billed for amounts that are different from the credit reservations made for those jobs, and different from what the users are expecting.
- Do not try to change a formula directly at the PBS complex, either by modifying the configuration file directly or by importing it. If the formula file for the Budgets hook is changed directly at the PBS complex, and not through the `amgr update` command, Budgets will throw an error.

3.3 Setting Budgets Configuration Attributes

Optionally, you can change the setting for the data_lifetime configuration attribute (it's not a parameter in the `am.conf` file; it's an attribute you set via `amgr update config`).

This attribute defines the maximum time allowed between updates to the value of a dynamic service unit. Budgets checks the update time for each dynamic service unit, for each project by calculating the value of (now - last update time for this dynamic service unit for this project).

If the value is not updated within the specified amount of time, Budgets logs a warning message in `/var/spool/am/<Budgets server hostname>.log`, but jobs can continue to run.

Format: integer seconds

Default: 3600

- To set the value of data_lifetime:

  amgr update config -n SU_DYNAMIC -V '{"data_lifetime":<new value>}'

  For example:

  ```
  amgr update config -n SU_DYNAMIC -V '{"data_lifetime":2400}'
  ```

  See section 4.2.3.8, “Updating Configuration Attributes”, on page 97.
- To see the value of data_lifetime:

  ```
  amgr ls config   -n data_lifetime
  ```

  See section 4.2.2.8, “Listing Budgets Configuration Attributes”, on page 88.

3.4 Configuring Budgets for Peer Scheduling

In a peer scheduling setup, different PBS complexes are set up to automatically run each others' jobs to dynamically load-balance jobs across the complexes. Budgets needs to be aware of all the PBS complexes in a peer scheduling environment.

To use Budgets when running jobs in multiple complexes in a peer scheduling environment:

1. Configure the Budgets hooks and its formula file at all of the PBS complexes involved in peer scheduling, and add one cluster to represent each PBS complex involved in peer scheduling to Budgets; see section 3.2, “Adding a PBS Complex and Setting its Billing Model”, on page 61.
2. Add the peer scheduling clusters to the project account or user account that will be running jobs, via `amgr update {user | project} -c <cluster>`; see section 4.2.3.3, “Updating Projects”, on page 93 and section 4.2.3.2, “Updating Users”, on page 93.

3.5 Changing Between Modes

3.5.1 Changing Mode from Postpaid to Prepaid

1. At each PBS complex, stop scheduling
2. Disable all PBS queues:
   ```
   qmgr -c 'set queue <queue name> enabled=false'
   ```
3. Allow all jobs to finish, or kill them
4. If necessary, reconcile all jobs:
   ```
   amgr reconcile [options]
   ```
   See section 4.3.6, “Reconciling Service Units”, on page 130
5. Stop Budgets:
   ```
   systemctl stop pbs_budget
   ```
6. Optionally make the balance zero for each account, period, and service unit
7. Change the AM_MODE configuration parameter in `am.conf` and set it to "prepaid"
8. Add the runjob hook event to am_hook:
   ```
   qmgr -c "set hook am_hook event += runjob"
   ```
9. Make sure that each account has sufficient credit to run their workload
10. Start Budgets
    ```
    systemctl start pbs_budget
    ```
11. Enable all PBS queues:
    ```
    qmgr -c 'set queue <queue name> enabled=true'
    ```
12. At each PBS complex, start scheduling

3.5.2 Changing Mode from Prepaid to Postpaid

1. At each PBS complex, stop scheduling
2. Disable all PBS queues:

   ```
   qmgr -c 'set queue <queue name> enabled=false'
   ```
3. Allow all jobs to finish, or kill them
4. If necessary, reconcile all jobs:

   ```
   amgr reconcile [options]
   ```

   See section 4.3.6, “Reconciling Service Units”, on page 130
5. Stop Budgets:

   ```
   systemctl stop pbs_budget
   ```
6. Optionally refund the balance for each account.
7. Change the AM_MODE configuration parameter in `am.conf` and set it to "postpaid"
8. Remove the runjob hook event from am_hook:

   ```
   qmgr -c "set hook am_hook event -= runjob"
   ```
9. Start Budgets

   ```
   systemctl start pbs_budget
   ```
10. Enable all PBS queues:

    ```
    qmgr -c 'set queue <queue name> enabled=true'
    ```
11. At each PBS complex, start scheduling

4 Budgets Commands

4.1 Budgets Commands

4.1.1 Command Path

To run Budgets commands, export the path of the `am` binaries to the PATH environment variable by using the command:

```
export PATH=$PATH:/opt/am/python/bin/
```

4.1.2 Using Budgets Commands

All Budgets commands are prefixed with "amgr ".

To see a list of Budgets subcommands with a single-line description for each command:

```
amgr <enter>
```

To get usage information for a command or subcommand:

<command> --help

<command> <subcommand> --help

For example:

`amgr add --help` provides information on how to use the main `amgr add` command

`amgr add period --help` provides information on how to use the `period` subcommand.

If you enter a command without the required arguments, Budgets will prompt you to enter them.

4.1.3 Tables of Budgets Commands

Table 4-1: Budgets Commands for Managing Elements

Function	Command	Element Subcommands	Required Privilege	Link
Adding elements	`amgr add`	`user, project, group, cluster, period, service-unit`	*admin*	Adding Elements
Listing elements	`amgr ls`	`user, project, group, cluster, period, service-unit, configuration, role, clouddata`	*user* can list periods, clusters, service units, own account, and roles, and all groups Member of project can list that project *manager* can list all elements *admin* can list cloud data	Listing Elements
Updating elements	`amgr update`	`user, project, group, cluster, period, service-unit, dynamicvalue, configuration, clouddata`	*admin*	Updating Elements
Removing elements	`amgr rm`	`user, project, group, cluster, period, service-unit`	*admin*	Removing Elements
Reporting elements	`amgr report`	`user, project, group, transaction`	*user* can get report on self and own jobs and transactions Project member can get report on that project *manager* or *investor* can get report on all groups and projects	Getting Reports on Elements
Applying limits to dynamic service units	`amgr limit`	`user, project`	Group *manager*	Applying Limits to Dynamic Service Units
Syncing formula file to cluster	`amgr sync`	`cluster`	*admin*	Syncing Formula File to PBS Complex

Table 4-2: Budgets Transaction and Account Checking Commands

Function	Command	Element Subcommands	Required Privilege	Link
Depositing service units	`amgr deposit`	`user, project, group`	*investor* for deposit to group Group *manager* for deposit to user or project account	Depositing Service Units
Checking balance of service units	`amgr check-balance`	`user, project, group`	*user* can check balance for self Project member can check balance for that project *manager* or *investor* can check balance for all groups and projects	Checking Service Unit Balance
Withdrawing service units	`amgr withdraw`	`user, project, group`	*investor* to withdraw from group Group *manager* to withdraw from user or project account	Withdrawing Service Units
Transferring service units	`amgr transfer`	`user, project, group`	*admin*	Transferring Service Units
Prechecking service unit balance	`amgr precheck`	`user, project, jobs`	*user* can precheck own balance *admin* or *teller* can check other balances	Prechecking Service Unit Balance
Acquiring service units	`amgr acquire`	`user, project`	*admin* or *teller*	Acquiring Service Units
Reconciling service units	`amgr reconcile`	`user, project`	*admin* or *teller*	Reconciling Service Units
Refunding service units	`amgr refund`	`transaction`	*admin*	Refunding Service Units

4.2 Commands for Managing Budgets Elements

You use these commands to add, remove, update, list, get reports on, apply limits to, and synchronize Budgets elements.

4.2.1 Adding Elements

amgr add {user | project | group | cluster | period | serviceunit}

4.2.1.1 Required Privilege

You must be *admin* to run this command.

4.2.1.2 Adding a User

4.2.1.2.i Synopsis

amgr add user -n <username> -A <accounting policy> -c <PBS server> -r <role> [-h <group list>] [-a <active>]

4.2.1.2.ii Description

Adds specified user to Budgets.

The specified user must already be a PBS complex user. Each user you add to Budgets should have an entry in the password file, with a password set, and a home directory on the Linux system where Budgets is installed.

When you add a user, you must assign a role, a cluster, and an accounting policy to that user.

4.2.1.2.iii Options

-n, --name <username>

String. Username to add.

The username and project name cannot be the same.

-A, --accounting-policy <accounting policy>

String. Specifies the accounting policy for the user account. Required.

Valid values: *begin_period* | *end_period* | *proportionate*

- A project with the begin_period accounting policy is charged in the period when the job begins.
- A project with the end_period accounting policy is charged in the period when the job ends.
- A project with the proportionate accounting policy is charged during the periods when the jobs were run. Each period is charged in proportion to the use in that period.

See section 1.7.4, “Accounting Policy”, on page 23.

-c, --clusters <PBS server>

String. Associates the user with the specified cluster. Required.

Use the `-c <PBS server>` option once for each cluster.

-r, --role <admin | investor | manager | teller | user>

String. Sets the role of the user. Required.

-h, --groups <group>

String. Associates the user account with specified groups.

Use the `-h <group>` option once for each group.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Activates or deactivates the user account.

Default: *True*, user account is active

4.2.1.2.iv Command Examples

Example 4-1: Add a user named "joe", setting the accounting policy to "begin_period", associating joe with cluster1 and group group01, and giving joe the role of *user*:

```
amgr add user -n joe -A begin_period -c cluster1 -r user -h group01
```

Example 4-2: Add a user to Budgets, then add them to group01 as a *manager*:

```
amgr add user -n user01 -A begin_period -c user1 -r manager -h group01
amgr update group -n group01 -M + user01
```

Example 4-3: Add user user1 to Budgets and add them to groups group01 and engineering as an *admin*, giving them the begin_period accounting policy, and adding them to the testbed1 and testbed2 PBS complexes:

```
amgr add user -n user1 -A begin_period -h group1 -h engineering -c testbed1 -c testbed2 -r admin
```

4.2.1.3 Adding a Project

4.2.1.3.i Synopsis

amgr add project -n <project name> -A <accounting policy> [-S <start date>] [-E <end date>] -c <PBS server> [-u <user>] [-h <group>] [-a <active>] [-m <metadata>]

4.2.1.3.ii Description

Adds specified project to Budgets.

4.2.1.3.iii Options

-n, --name <project name>

String. Name of project to add.

The project name cannot be the same as a username.

-A, --accounting-policy <accounting policy>

String. Specifies the accounting policy for the project account. Required.

Valid values: *begin_period* | *end_period* | *proportionate*

- A project with the begin_period accounting policy is charged in the period when the job begins.
- A project with the end_period accounting policy is charged in the period when the job ends.
- A project with the proportionate accounting policy is charged during the periods when the jobs were run. Each period is charged in proportion to the use in that period.

See section 1.7.4, “Accounting Policy”, on page 23.

-S, --start-date <start date>

Date. Start date of the project. Optional.

Format: YYYY-MM-DD

-E, --end-date <end date>

Date. End date of the project. Optional.

Format: YYYY-MM-DD

-c, --clusters <PBS server>

String. Associates the specified cluster with the project account.

Use the `-c <PBS server>` option once for each cluster. To add multiple clusters:

-c <cluster1> -c <cluster2> ... -c <clusterN>

-u, --users <username>

String. Associates the specified user with the project account.

Use the `-u <username>` option once for each user. To add multiple users:

-u <user1> -u <user2> ... -u <userN>

-h, --groups <group name>

String. Associates the specified group with the project.

Use the `-h <group name>` option once for each group. To add multiple groups:

-h <group1> -h <group2> ... -h <groupN>

-a, --active <True|TRUE|true|t|1|False|FALSE|false|f|0>

Boolean. Sets the project to active or inactive.

-m, --metadata <metadata>

String. Modifies metadata attribute for the project.

Format: comma-separated key-value pairs

Syntax:

<key1>:<value1>,<key2>:<value2>...<keyN>:<valueN>

Keys are undefined.

4.2.1.3.iv Example

Example 4-4: Add a project named proj1 and give it metadata consisting of type:weather and region:asia:

```
amgr add project -n proj1 -A begin_period -S 2022-01-02 -E 2022-28-02 -c cluster1 -m
    type:weather,region:asia
```

4.2.1.4 Adding a Group

4.2.1.4.i Synopsis

amgr add group -n <group name> [-I <investor>] [-M <manager>] [-a <active>]

4.2.1.4.ii Description

Define a group and add it to Budgets.

4.2.1.4.iii Options

-n, --name <group name>

String. Group to add.

-I, --investors <investor username>

String. Associates this investor with the specified group.

This username must already exist in Budgets and have the *investor* or *admin* role.

You can associate one investor per `-I` option, and you can associate multiple investors per command line. To associate multiple investors:

-I <investor1> -I <investor2> ... -I <investorN>

-M, --managers <manager username>

String. Associates this manager with the specified group.

This username must already exist in Budgets and have the *manager*, *investor*, or *admin* role.

You can associate one manager per -M option, and you can associate multiple managers per command line. To associate multiple managers:

-M <manager1> -M <manager2> ... -M <managerN>

Group managers can transfer funds from the group to projects and user accounts.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Sets the group active or inactive.

Default: active

4.2.1.5 Adding a Cluster

4.2.1.5.i Synopsis

amgr add cluster -n <PBS server> [-f <billing formula filename>] [-a <active>]

4.2.1.5.ii Description

Adds a cluster data structure to represent the specified PBS complex.

You can specify the billing formulas used for each cluster at the time you add the cluster, or later; although if the formula file at the cluster does not match the formula file at the complex, jobs cannot run at that complex. When you specify the billing formulas, this updates the cluster data structure with the formulas.

You can set the cluster to active or inactive.

You can add multiple PBS complexes, but only one per command line.

4.2.1.5.iii Options

-n, --name <PBS server>

String. Name of the PBS server for the PBS complex.

-f, --formula <formula filename>

Sets the formula file at the cluster. Make sure these are the same formulas as the ones at the complex.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Sets cluster active or inactive. When the cluster is active, it can run jobs.

Default: active

4.2.1.6 Adding a Period

4.2.1.6.i Synopsis

amgr add period -n <period name> -S <start date> -E <end date> [-p <name of parent period>]

4.2.1.6.ii Description

Adds the specified period.

Make sure that periods at the same level do not overlap. For example, if Quarter1 ends March 31st, make sure that Quarter2 does not begin sooner than April 1st.

If you want to create periods with a parent-child relationship, you must create the parent period first. You cannot add a parent to an existing child. For example, if you want Year as the parent and Quarter1, Quarter2, etc., as children, create Year first.

If you create child periods, make sure that they fit within the parent period.

Make sure that your period hierarchy is finalized BEFORE doing any transactions or running any jobs; you cannot update or remove periods once transactions have been performed or jobs have started.

See section 1.7.1, "Periods, Allocation Periods, Billing Periods", on page 18.

4.2.1.6.iii Options

-n, --name <period name>

String. Period to add.

-S, --start-date <start date>

Date. Start date of the period.

Format: YYYY-MM-DD

-E, --end-date <end date>

Date. End date of the period.

Format: YYYY-MM-DD

-p, --parent <name of parent period>

String. Specifies the parent period. Optional.

4.2.1.7 Adding a Service Unit

4.2.1.7.i Synopsis

amgr add serviceunit -n <service unit name> [-t <type>] [-a <active>] [-d <description>]

4.2.1.7.ii Description

Adds a service unit to Budgets. This service unit is defined in the formula file.

Add a standard service unit to measure consumption of a resource internally tracked by PBS, for example CPU hours.

Add a dynamic service unit to define a quota for an external resource.

Default type is standard service unit (SU_STANDARD).

4.2.1.7.iii Options

-n, --name <service unit name>

String. Name of service unit to add. If this is a standard service unit, name must match name used in formula file.

Blank spaces are not allowed.

-t, --type {SU_STANDARD | SU_DYNAMIC}

String. Specifies type for this service unit.

Default: *SU_STANDARD*

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Sets service unit active or inactive.

-d, --description <service unit description>

String. Describes the service unit.

Can contain alphanumeric and any special characters except double quotes.

If the description contains anything except alphanumeric, enclose it in double quotes.

4.2.1.7.iv Command Examples

Example 4-5: Adding a standard service unit named cpu_hrs to be used for CPU hours:

```
amgr add serviceunit -n cpu_hrs -d "CPU hours"
```

Example 4-6: Adding a dynamic service unit named luster:

```
amgr add serviceunit -n luster -t SU_DYNAMIC
```

4.2.2 Listing Elements

amgr ls {user | period | project | cluster | serviceunit | role | clouddata}

Prints out Budgets elements. You can list only elements that have already been added to Budgets via `amgr add` or `amgr update`.

By default, this command lists only active elements. To list active elements:

```
amgr ls
```

To list inactive elements, use the `–a False` option. For example, to list all the clusters which are inactive:

```
amgr ls cluster -a False
```

Use the `–l` switch for `amgr ls` commands for more detailed information.

Use the `–j` switch for `amgr ls` commands to get the detailed information output in JSON format. This output can be used by other programs which can process JSON format.

4.2.2.1 Required Privilege

A *user* can list periods, clusters, service units, own account, own projects, own role, and all groups.

A member of a project can list that project.

A *manager* can list all elements.

4.2.2.2 Listing Users

4.2.2.2.i Synopsis

amgr ls user [-n <username] [-a <active>] [-h <group name>] [-l | -j] [-r <role>]

4.2.2.2.ii Description

When used with no options, lists all the users that exist in Budgets. When you specify a username, prints information about that user.

4.2.2.2.iii Options

-n, --name <username>

String. Specifies the name of the user.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters users by active status. You can list either active or inactive users, but not both.

Default: active

-h, --group <group name>

String. Lists all users in specified group.

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

-r, --role <role>

String. Lists all users with specified role.

4.2.2.3 Listing Projects

4.2.2.3.i Synopsis

amgr ls project [-n <project name] [-a <active>] [-h <group name>] [-l | -j] [-u <username>]

4.2.2.3.ii Description

When used with no options, lists all the projects that exist in Budgets. When you specify a project name, prints information about that project.

4.2.2.3.iii Options

-n, --name <project name>

String. Specifies the name of the project.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters projects by active status. You can list either active or inactive projects, but not both.

Default: active

-h, --group <group name>

String. Lists all projects associated with specified group.

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

-u, --user <username>

String. Lists all projects with which specified user is associated.

4.2.2.4 Listing Groups

4.2.2.4.i Synopsis

amgr ls group [-n <group name>] [-a <active>] [-I <investor name>] [-l | -j] [-M <manager name>]

4.2.2.4.ii Description

When used with no options, lists all the groups that exist in Budgets. When you specify a group name, prints information about that group.

4.2.2.4.iii Options

-n, --name <group name>

String. Specifies the name of the group.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters groups by active status. You can list either active or inactive groups, but not both.

Default: active

-I, --investor <investor name>

String. Lists all groups associated with specified investor.

-j, --json-info

Display information in JSON format.

-l, --list-info

Display information in list format.

-M, --manager <manager name>

String. Lists all groups associated with specified manager.

4.2.2.5 Listing Clusters

4.2.2.5.i Synopsis

amgr ls cluster [-n <PBS server>] [-a <active>] [-f] [-l | -j]

4.2.2.5.ii Description

By default, lists all active clusters. You can specify whether you want to see active or inactive clusters. To display inactive clusters:

```
amgr ls cluster -a False
```

4.2.2.5.iii Options

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters clusters by active status. You can list either active or inactive clusters, but not both.

Default: active

-n, --name <PBS server>

String. Name of the cluster to list.

You can list one cluster per command line.

-f, --formula

Prints formula file to `/tmp`, and prints the path to that file on screen.

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

4.2.2.5.iv Examples

Example 4-7: To print the formula file for cluster1:

```
amgr ls cluster -n cluster1 -f
```

Output formula file: `/tmp/cluster1_formula.json`

4.2.2.6 Listing Periods

4.2.2.6.i Synopsis

amgr ls period [-n <period>] [-l | -j]

4.2.2.6.ii Description

Prints out information about periods.

By default, this lists all periods that exist in the hierarchy.

See section 1.7.1, “Periods, Allocation Periods, Billing Periods”, on page 18.

4.2.2.6.iii Options

-n, --name <period name>

String. Name of the period to list.

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

4.2.2.7 Listing Service Units

4.2.2.7.i Synopsis

amgr ls serviceunit [-n <service unit name] [-a <active>] [-l | -j]

4.2.2.7.ii Description

When used with no options, lists all the service units that exist in Budgets. When you specify a service unit name, prints information about that service unit.

4.2.2.7.iii Options

-n, --name <service unit name>

String. Specifies the name of the service unit.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters service units by active status. You can list either active or inactive service units, but not both.

Default: active

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

4.2.2.8 Listing Budgets Configuration Attributes

4.2.2.8.i Synopsis

amgr ls config [-n <attribute name>] [-l | -j]

4.2.2.8.ii Description

Prints a list of the Budgets configuration attributes. See section 3.3, "Setting Budgets Configuration Attributes", on page 73.

4.2.2.8.iii Options

-n, --name <attribute name>

String. Specifies the name of the configuration attribute.

-l, --list-info

Display information in list format.

-j, --json-info

Display information in JSON format.

4.2.2.8.iv Sample Output

Sample output for `amgr ls config`:

```
SU_DYNAMIC
configuration = {'data_lifetime': 3600}
id = 1
created_user_name = root
created_date = 2022-05-12 09:42:53.312722+05:30
last_updated_user_name = root
last_updated_date = 2022-05-14 15:55:54.339+05:30
```

4.2.2.9 Listing Roles

4.2.2.9.i Synopsis

amgr ls role [-n <role>] [-a <active>] [-l | -j]

4.2.2.9.ii Description

Prints whether the role exists.

Every user in Budgets must have an assigned role. See section 1.4, "Roles", on page 7.

Roles in Budgets:

admin

Can perform all operations.

investor

Can deposit and withdraw service units to and from groups with which the investor is associated.

manager

Can deposit and withdraw service units to and from projects and users that are associated with groups with which the manager is associated.

teller

Special role for performing automated acquire and reconcile transactions on behalf of users.

user

Assigned to one or more projects; can run jobs using user budget or project budget.

4.2.2.9.iii Options

-n, --name <role>

String. Name of the role to list.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Filters roles by active status. You can list either active or inactive roles, but not both.

Default: active

-l, --list-info

Display information in list format. Information includes whether the role is active for each user with the role.

-j, --json-info

Display information in JSON format.

4.2.2.9.iv Command Example

Example 4-8: To show information for users with *admin* role:

```
amgr ls role -n admin
```

Example 4-9: To show more information about users with *admin* role, we use the -l switch:

```
amgr ls role -n admin -l
```

Example 4-10: To show information about users with *admin* role in JSON format, we use the -j switch:

```
amgr ls role -n admin -j
```

4.2.2.10 Listing Cloud Data

4.2.2.10.i Synopsis

amgr ls clouddata [-n <cloud account name] [-t <instance type>] [-l | -j]

4.2.2.10.ii Description

To see a list of all cloud accounts and instance types, but not cost data, use no options.

To see a list of all cloud accounts and instance types, along with instance type costs, use either the `-l` or `-j` options, and use neither the `-n` nor the `-t` options.

To see a list of instance types associated with a cloud account, use the `-n` option to specify that account.

To see a list of cloud accounts associated with an instance type, use the `-t` option to specify that instance type.

4.2.2.10.iii Options

-n, --account-name <cloud account name>

String. Specifies the name of the cloud account.

Lists the instance types associated with this cloud account.

-t, --instance-type <instance type>

String. Specifies the instance type.

Lists the cloud accounts associated with this instance type.

(neither -n nor -t)

With `-l` option: lists all cloud accounts and associated instance types, along with cost information.

With `-j` option: prints all cloud accounts and associated instance types, along with cost information, in JSON format.

With neither the `-l` or `-j` options, lists all cloud accounts and associated instance types, without cost information.

-l, --list-info

Displays information in list format.

-j, --json-info

Displays information in JSON format.

4.2.2.10.iv Examples

Example 4-11: List all cloud accounts and their associated instance types:

```
amgr ls clouddata
xyz-aws:c4.large
xyz-azure:t2.micro
```

Example 4-12: List the instance types associated with a specific cloud account:

```
amgr ls clouddata -n xyz-aws
xyz-aws:c4.large
```

Example 4-13: List the cloud accounts associated with a specific instance type:

```
amgr ls clouddata -t t2.micro
xyz-azure:t2.micro
```

Example 4-14: Validate that a particular cloud account is associated with a particular instance type:

```
amgr ls clouddata -n xyz-aws -t c4.large
xyz-aws:c4.large
```

Example 4-15: List all cloud accounts and associated instance types along with cost information for each instance type:

```
amgr ls clouddata -l
xyz-aws
    instance_type = c4.large
    cost_info = {'cost': 100, 'ncpus': 20, 'overhead': 40, 'unit': 'hour'}
    metadata = {'provider': 'aws', 'scenario': 'xyz-aws-poc'}
    id = 2
    created_user_name = root
    created_date = 2022-05-18 17:33:00.310976+05:30
    last_updated_user_name = root
    last_updated_date = 2022-05-18 17:33:00.310976+05:30

xyz-azure
    instance_type = t2.micro
    cost_info = {'cost': 80, 'ncpus': 10, 'overhead': 20, 'unit': 'hour'}
    metadata = {'provider': 'azure', 'scenario': 'xyz-azure-poc'}
    id = 3
    created_user_name = root
    created_date = 2022-05-18 17:33:00.310976+05:30
    last_updated_user_name = root
    last_updated_date = 2022-05-18 17:33:00.310976+05:30
```

4.2.3 Updating Elements

amgr update {user | period | project | group | cluster | serviceunit | clouddata}

Updates the specified element.

You cannot change the name of an element once it is created.

4.2.3.1 Required Privilege

You must be *admin* to run this command.

4.2.3.2 Updating Users

4.2.3.2.i Synopsis

amgr update user -n <username> [-A <accounting policy>] [-c <cluster list>] [-r <roles>] [-h <group list>] [-a <active>]

4.2.3.2.ii Description

Update information for a user.

4.2.3.2.iii Options

-n, --name <username>

String. Name of user account to update.

-A, --accounting-policy <begin_period | end_period | proportionate>

Specifies the accounting policy for the user account. It can be begin_period, end-period, or proportionate

-c, --clusters <cluster update>

String. Specifies clusters to link to or unlink from the user.

To link or unlink clusters, use the + or - operator followed by a comma-separated list of clusters. Spaces are not allowed. Use a separate `amgr update user -n <username> -c <cluster update>` command for each + or - operation.

For example, to link clusters:

```
amgr update user -n user1 -c + cluster1,cluster2,cluster3
```

Or to unlink clusters:

```
amgr update user -n user1 -c - cluster4,cluster5,cluster6
```

-r, --roles <role>

Sets the role of the user.

-h, --groups <group update>

String. Specifies groups to link to or unlink from the user.

To link or unlink groups, use the + or - operator followed by a comma-separated list of groups. Spaces are not allowed. Use a separate `amgr update user -n <username> -h <group update>` command for each + or - operation.

For example, to link groups:

```
amgr update user -n user1 -h + group1,group2,group3
```

Or to unlink groups:

```
amgr update user -n user1 -h - group4,group5,group6
```

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Set the project to active or inactive.

4.2.3.3 Updating Projects

4.2.3.3.i Synopsis

amgr update project -n <project name> [-A <accounting policy>] [-S <start date>] [-E <end date>] [-c <cluster update>] [-u <user update>] [-h <group update>] [-a <active>] [-m <metadata update>]

4.2.3.3.ii Description

Updates project information. You can set new values for existing elements, add new elements such as users, groups, or metadata, and you can remove elements. Use the + operator to add or update elements and the - operator to remove elements.

4.2.3.3.iii Options

-n, --name <project name>

String. Name of the project to update.

-A, --accounting-policy <accounting policy>

String. Specifies the accounting policy for the project.

Value is one of *begin_period*, *end_period,* or *proportionate*.

-S, --start-date <start date>

Start date of the project.

Format: YYYY-MM-DD

-E, --end-date <end date>

End date of the project.

Format: YYYY-MM-DD

-c, --clusters <cluster update>

String. Specifies clusters to link to or unlink from the project.

To link or unlink clusters, use the + or - operator followed by a comma-separated list of clusters. Spaces are not allowed. Use a separate `amgr update project -n <project name> -c <cluster update>` command for each + or - operation.

For example, to link clusters:

```
amgr update project -n MyProject -c + cluster1,cluster2,cluster3
```

Or to unlink clusters:

```
amgr update project -n MyProject -c - cluster4,cluster5,cluster6
```

-u, --users <user update>

String. Specifies users to link to or unlink from the project.

To link or unlink users, use the + or - operator followed by a comma-separated list of users. Spaces are not allowed. Use a separate `amgr update project -n <project name> -u <user update>` command for each + or - operation.

For example, to link users:

```
amgr update project -n MyProject -u + user1,user2,user3
```

Or to unlink users:

```
amgr update project -n MyProject -u - user4,user5,user6
```

-h, --groups <group update>

String. Specifies groups to link to or unlink from the project.

To link or unlink groups, use the + or - operator followed by a comma-separated list of groups. Spaces are not allowed. Use a separate `amgr update project -n <project name> -h <group update>` command for each + or - operation.

For example, to link groups:

```
amgr update project -n MyProject -h + group1,group2,group3
```

Or to unlink groups:

```
amgr update project -n MyProject -h - group4,group5,group6
```

-m, --metadata <metadata update>

Adds, updates, or removes metadata from the project.

To add metadata: + operator followed by a comma-separated list of key-value pairs. Spaces are not allowed. For example:

```
amgr update project -n MyProject -m + <key1>:<value>,<key2>:<value2>
```

To remove metadata: - operator followed by a comma-separated list of keys only. Spaces are not allowed. For example:

```
amgr update project -n MyProject -m - <key3>,<key4>
```

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Sets the project to active or inactive.

4.2.3.3.iv Command Example

Add a new metadata element named "currency", set it to "rupee", and update the existing data region to "america":

```
amgr update project -n proj1 -m + currency:rupee,region:america
```

Remove existing metadata elements "type" and "region":

```
amgr update project -n proj1 -m - type,region
```

Multiple operations in single command:

```
amgr update project -n proj1 -m + currency:dollar,"Lead Researcher":"Jane Smith" -m - type,region
```

4.2.3.4 Updating Groups

4.2.3.4.i Synopsis

amgr update group -n <group name> [-I <investors>] [-M <managers>] [-a <active>]

4.2.3.4.ii Description

Update information for a group.

4.2.3.4.iii Options

-n, --name <group name>

String. Name of the group to update.

-I, --investors <investor update>

String. Specifies investors to link to or unlink from the user.

To link or unlink investors, use the + or - operator followed by a comma-separated list of investors. Spaces are not allowed. Use a separate `amgr update group -n <group name> -I <investor update>` command for each + or - operation.

For example, to link investors:

```
amgr update group -n group1 -I + investor1,investor2,investor3
```

Or to unlink investors:

```
amgr update group -n group1 -I - investor4,investor5,investor6
```

-M, --managers <manager update>

String. Specifies managers to link to or unlink from the user.

To link or unlink managers, use the + or - operator followed by a comma-separated list of managers. Spaces are not allowed. Use a separate `amgr update group -n <group name> -M <manager update>` command for each + or - operation.

For example, to link managers:

```
amgr update group -n group1 -M + manager1,manager2,manager3
```

Or to unlink managers:

```
amgr update group -n group1 -M - manager4,manager5,manager6
```

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Sets the group to be active or inactive.

4.2.3.5 Updating Clusters

4.2.3.5.i Synopsis

amgr update cluster -n <PBS server> [-a <active>] [-f <formula filename>]

4.2.3.5.ii Description

Updates the cluster data structure representing a PBS complex with the specified formula file. Pushes the formula file to the PBS complex. Imports the formula file (the Budgets hook configuration file) into both Budgets hooks at the complex. Updates the database with the new formulas for this cluster.

If the formulas are different in the PBS hook (the .CF file) and the cluster data structure, jobs cannot run.

4.2.3.5.iii Options

-n, --name <PBS server>

String. Name of the cluster to update.

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Sets the cluster active or inactive.

Default: active

-f, --formula <formula filename>

String. Updates the specified formula file at the specified cluster data structure.

4.2.3.6 Updating a Period

4.2.3.6.i Synopsis

amgr update period -n <period name> [-S <start date>] [-E <end date>]

4.2.3.6.ii Description

Updates the specified period.

You cannot update a period that has any associated jobs or transactions.

See section 1.7.1, “Periods, Allocation Periods, Billing Periods”, on page 18.

4.2.3.6.iii Options

-n, --name <period name>

String. Period to update.

-S, --start-date <start date>

Date. Start date of the period.

Format: YYYY-MM-DD

-E, --end-date <end date>

Date. End date of the period.

Format: YYYY-MM-DD

4.2.3.7 Updating a Service Unit

4.2.3.7.i Synopsis

amgr update serviceunit -n <service unit name> [-t <type>] [-a <active>] [-d <description>]

4.2.3.7.ii Description

Updates a service unit.

You can change the type of a service unit, but there are restrictions:

- You can change standard to dynamic only when no transactions have taken place
- You can change dynamic to standard only when no updates have been made to the external dynamicvalue

4.2.3.7.iii Options

-n, --name <service unit name>

String. Name of service unit to update.

Blank spaces are not allowed.

-t, --type {SU_STANDARD | SU_DYNAMIC}

String. Specifies type for this service unit.

Default: *SU_STANDARD*

-a, --active {True|TRUE|true|t|1|False|FALSE|false|f|0}

Boolean. Sets service unit active or inactive.

-d, --description <service unit description>

String. Describes the service unit.

Can contain alphanumeric and any special characters except double quotes.

If the description contains anything except alphanumeric, enclose it in double quotes.

4.2.3.8 Updating Configuration Attributes

4.2.3.8.i Synopsis

amgr update config -n <attribute name> -V <update string>

4.2.3.8.ii Description

Update Budgets configuration attributes. See section 3.3, “Setting Budgets Configuration Attributes”, on page 73.

Updates one attribute at a time.

4.2.3.8.iii Options

-n, --name <attribute name>

String. Name of the Budgets configuration attribute to update.

-V, --config-value <update string>

JSON formatted string of configuration key-value pairs.

Format:

amgr update config -n SU_DYNAMIC -V '{"<attribute name>": <value>}'

Example:

```
amgr update config -n SU_DYNAMIC -V '{"data_lifetime": 1000}'
```

4.2.3.8.iv Examples

```
amgr update config -n SU_DYNAMIC -V '{"data_lifetime": 3610}'
```

4.2.3.9 Updating Dynamic Service Unit Usage

4.2.3.9.i Synopsis

amgr update dynamicvalues -v <update specification>

4.2.3.9.ii Description

Reports updated value of the specified dynamic service unit to Budgets. Value is for usage by a project or a user.

At a given point in time, a dynamic service unit is a snapshot of the current usage of a particular external resource such as scratch. This usage is reported by an external script that you write. The external script is typically a `cron` job that calls `amgr update dynamicvalues`, which updates the value of the dynamic service unit.

Budgets compares the reported value with the specified quotas, and allows jobs to run as long as the usage is below the quota.

4.2.3.9.iii Required Privilege

You must be *admin* to run this command.

4.2.3.9.iv Options

-v, --value <update specification>

String in JSON format.

Format to update usage for a user:

'{"<dynamic service unit>": {"<username>":{"total":<total>}}}'

Format to update total usage for a project:

'{"<dynamic service unit>": {"<project>":{"total":<total>}}}'

You must include the keyword "total".

4.2.3.9.v Examples

Example 4-16: Set user user1's consumption of storage to 8:

```
amgr update dynamicvalues -v '{"storage": {"user1": {"total":8}}}'
```

Example 4-17: Set project project1's consumption of storage to 30:

```
amgr update dynamicvalues -v '{"storage": {"project1": {"total":30}}}'
```

4.2.3.10 Updating Cloud Cost Data

4.2.3.10.i Synopsis

amgr update clouddata [-v <update specification> | -J <input file>]

4.2.3.10.ii Description

Updates Budgets with cloud cost data for specified accounts and instances. The administrator gets the cloud cost data from the vendor, then uses this command to update Budgets. The command transmits the updated cost information via either a JSON string or a JSON file.

Budgets uses these costs when computing service unit charges.

The cost to use an instance type depends on service provider charges, which may vary according to the instance type. Spot instances may vary more often than others.

This command is intended to be used in a `cron` job or a periodic hook.

4.2.3.10.iii Required Privilege

You must be *admin* to run this command.

4.2.3.10.iv Options

-v, --value <update specification>

String in JSON format. You populate this string with data collected from service provider(s). Format:

```
'{
    "<cloud account 1>":{
        "<instance type 1>":{
            "cost_info":{
                "cost":<cost per unit time>,
                "ncpus":<CPUs in one instance>,
                "overhead":<cost for overhead>,
                "unit":<unit of time>
            },
            "metadata":{
                "key1":"value1",
                "key2":"value2",
                ...
            }
        }
    },
    "<cloud_account2>":{
        "<instance_type2>":{...}
    },
    ...
}'
```

Cost information section contains the following:

cost
: Float. Instance cost per unit of time.

ncpus
: Integer. Total number of CPUs available inside one instance.

overhead
: Float. Total overhead cost for one instance.

unit
: String. Unit of time for the *cost* and *overhead* values.

Allowed values: *hour* | *minute* | *second*

Metadata section is key-value pairs containing metadata for the instance.

Format:

"<key>":"<value>","<key>":"<value>", ...

Maximum of 10 pairs allowed.

Cannot be used with `-J` option.

-J, --json-file <path to input file>

String. Path to input file containing cost data. Can be absolute or relative to directory where command is run.

Format: same as JSON string argument to `-v <update specification>` option.

Cannot be used with `-v` option.

--help

Prints usage information and exits.

4.2.3.10.v Examples

Example 4-18: Update cloud cost data for the cloud account named "CloudAccount1" and the instance type "Standard_D2s_v3". We show how to update using a JSON string or an input file:

Update via JSON string:

```
amgr update clouddata -v
    '{"CloudAccount1":{"Standard_D2s_v3":{"cost_info":{"cost":0.105,"ncpus":2,"overhead":0.02,"u
    nit":"hour"},"metadata":{"provider":"azure","scenario":"azureScenario1"}}}}'
```

Update via input file:

```
amgr update clouddata -J /tmp/my_cloud_cost_data.json
```

Where:

```
cat /tmp/my_cloud_cost_data.json
{
    "CloudAccount1":{
        "Standard_D2s_v3":{
            "cost_info":{
                "cost":0.105,"ncpus":2,"overhead":0.02,"unit":"hour"
            },
            "metadata":{
                "provider":"azure","scenario":"azureScenario1"
            }
        }
    }
}
```

Example 4-19: Update cloud cost data for two separate cloud accounts, using a file:

```
amgr update clouddata -J /tmp/cloud_cost_data.json
```

Where:

```
cat /tmp/cloud_data.json
{
    "CloudAccount1":{
        "Standard_D2s_v3":{
            "cost_info":{"cost":0.105,"ncpus":2,"overhead":0.02,"unit":"hour"},
            "metadata":{"provider":"azure","scenario":"azureScenario"}
        }
    }
    "CloudAccount2":{
        "Standard_D2s_v4":{
            "cost_info":{"cost":0.205,"ncpus":2,"overhead":0.05,"unit":"hour"}
        }
    }
}
```

4.2.3.10.vi Caveats for Updating Cloud Cost Data

Keep in mind that the units you report to Budgets using this command will affect how you use units in the formula file. In the formula file, Budgets assumes that cost units are in seconds.

4.2.4 Removing Elements

amgr rm {user | project | group | cluster | period | serviceunit}

4.2.4.1 Required Privilege

You must be *admin* to run this command.

4.2.4.2 Removing a User

4.2.4.2.i Synopsis

amgr rm user -n <username>

4.2.4.2.ii Description

Removes specified user from Budgets, unless the user has any past or current associated jobs or transactions, in which case the user is made inactive.

4.2.4.2.iii Options

-n, --name <username>

String. Name of the user to remove.

4.2.4.2.iv Command Examples

Example 4-20: Remove a user named "joe":

```
amgr rm user -n joe
```

4.2.4.3 Removing a Project

4.2.4.3.i Synopsis

amgr rm project -n <project name>

4.2.4.3.ii Description

Removes specified project from Budgets, unless the project has any past or current associated jobs or transactions, in which case the project is made inactive.

4.2.4.3.iii Options

-n, --name <project name>

String. Name of the project to remove.

The project name cannot be the same as a username.

4.2.4.3.iv Example

Example 4-21: Remove a project named proj1:

```
amgr rm project -n proj1
```

4.2.4.4 Removing a Group

4.2.4.4.i Synopsis

amgr rm group -n <group name>

4.2.4.4.ii Description

Removes specified group from Budgets, unless the group has any past or current associated transactions, in which case the group is made inactive.

4.2.4.4.iii Options

-n, --name <group name>

String. Name of the group to remove.

4.2.4.5 Removing a Cluster

4.2.4.5.i Synopsis

amgr rm cluster -n <PBS server>

4.2.4.5.ii Description

Removes the specified cluster from Budgets, unless the cluster has any past or current associated jobs or transactions, in which case the cluster is made inactive.

4.2.4.5.iii Options

-n, --name <PBS server>

String. Name of the cluster to remove.

4.2.4.6 Removing a Period

4.2.4.6.i Synopsis

amgr rm period -n <period name>

4.2.4.6.ii Description

Removes specified period from Budgets, if the period and all parent periods have no associated jobs or transactions. If either the period or a parent has any associated jobs or transactions, the period is made inactive.

See section 1.7.1, "Periods, Allocation Periods, Billing Periods", on page 18.

4.2.4.6.iii Options

-n, --name <period name>

String. Name of the period to remove.

4.2.4.7 Removing a Service Unit

4.2.4.7.i Synopsis

amgr rm serviceunit -n <service unit name>

4.2.4.7.ii Description

Removes specified service unit from Budgets, unless the service unit has any past or current associated transactions, in which case the service unit is made inactive.

4.2.4.7.iii Options

-n, --name <service unit name>

String. Name of service unit to remove.

If this is a standard service unit, the name must match name used in formula file.

Blank spaces are not allowed.

If this is a standard service unit, you must also remove this service unit from all formulas, otherwise jobs will not run.

4.2.4.7.iv Command Examples

Example 4-22: Removing a standard service unit named cpu_hrs:

```
amgr rm serviceunit -n cpu_hrs
```

Example 4-23: Removing a dynamic service unit named luster:

```
amgr rm serviceunit -n luster
```

4.2.5 Getting Reports on Elements

The `amgr report` command allows you to get reports for projects, groups, jobs, transactions, and users.

amgr report {user | project | group | transaction}

Use the `-l` option to the `amgr report` commands for more detailed information.

4.2.5.1 Required Privilege

A *user* can get a report on their own usage, jobs, and transactions, and on any project account with which the user is associated.

A *manager* can get reports on users and projects that are associated with any groups with which the manager is associated.

A member of a project can get a report on that project.

An *admin* or *investor* can get a report on any group and project.

4.2.5.2 Getting User Reports

4.2.5.2.i Synopsis

amgr report user -n <username> [-s <service unit name> | -t <service unit type>] [-h] [-p <period>] [-S <start date>] [-E <end date>] [-l] [-o <output file>] [-r] [-A]

4.2.5.2.ii Description

By default, this prints information for all standard service units for the current period that is lowest in the hierarchy (shortest time division). You can refine your output by specifying service unit name and type.

By default, this prints the report to the screen in human-readable tables. You can print to a file, and you can print the report in raw (CSV) format.

4.2.5.2.iii Options

-n, --name <username>

String. Username on which to get report.

-s, --serviceunit <service unit name>

Prints report for specified service unit.

If you do not specify a service unit, the report includes all service units of the specified type. If you do not specify type, it is *SU_STANDARD*.

-t, --sunit-type <service unit type>

Use this option to see dynamic service units. By default, this command prints only standard service units.

Cannot be used with `-l`, `-g`, or `-h` options.

Type can be one of *SU_STANDARD* or *SU_DYNAMIC*.

Default: *SU_STANDARD*

-g, --global-view

Lists project details and a summary of all transactions with a detailed listing of each transaction in JSON format.

Cannot be used with `-t SU_DYNAMIC`.

-h, --stakeholder-info

Lists groups which have invested in the user and their invested amounts.

Cannot be used with `-t SU_DYNAMIC`.

-p, --period <period name>

Specifies report period.

Cannot be used with `-S` and `-E` options.

Default: current period that is lowest in hierarchy (shortest time division)

-S, --start-date <start date>

Reports activity starting at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-E, --end-date <end date>

Reports activity ending at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-l, --list-info

Lists all transactions that have happened for the specified user.

Use the -l switch to get the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type, serviceunit, amount, balance, reconciled, period, am_mode, and comment, in columns.

Cannot be used with `-t SU_DYNAMIC`.

-o, --out-file <output filename>

Budgets prints the report in a human-readable form to the specified output file.

Without this option, Budgets prints the report to the screen.

-r, --raw-output

Print report in raw (CSV) form with no padding. This form is easier to parse but harder for a human to read.

-A, --allocated-view

For listing investor information. Lists remaining credit allocated from this investor to each group in which the investor invested.

4.2.5.2.iv Output Format

Output for the report contains the Name, Service Unit, Period, Opening Balance, Total Credits, Total Debits, Total Debits (Reconciled), Total Debits (Authorized), and Net Balance.

Opening Balance

Opening balance of user account on specified start date

Total Credits

Sum of deposits; does not decrease from transfers or withdrawals or usage. Associated with a period.

Total Debits (Reconciled)

Total consumed amount for a period.

Total Debits (Authorized)

Amount held by running jobs plus amount held by unreconciled finished jobs, for a period.

Current balance

Opening Balance + Total Credits - Total Debits - Total Debits (Reconciled) - Total Debits (Authorized). For period.

4.2.5.2.v Command Examples in Postpaid Mode

Example 4-24: User report:

```
amgr report user -n centos
----------------------------------------------------------------------
name     | period    | serviceunit    | total_outstanding   | metadata
----------------------------------------------------------------------
centos   | 2022      | cpu_hrs        | 480.0               | None
```

Example 4-25: User report in long format:

```
amgr report user -n centos -l
```

4.2.5.2.vi Command Examples in Prepaid Mode

Example 4-26: User report in long format:

amgr report user -n <username> -s <service unit> -S <start date> -E <end date> -l

Example 4-27: To report user storage usage:

```
amgr report user -n user1 -t SU_DYNAMIC
-------------------------------------------------------------------------------------
name  | serviceunit | period   | limit | last_reported_time             | total_consumed
-------------------------------------------------------------------------------------
user1 | storage     | 2022.feb| 12.0  | 2022-02-11 18:58:09.48498+00:00| 8.0
```

Example 4-28: User report showing dynamic service units:

```
amgr report user -n pbsuser -t SU_DYNAMIC

----------------------------------------
name    | serviceunit | period | limit
----------------------------------------
pbsuser | luster      | 2022   | 600.0
pbsuser | scratch     | 2022   | 800.0

---------------------------------------------------------
| last_reported_time                | total_consumed
---------------------------------------------------------
| 2022-04-07 12:40:22.470078+05:30 | 40
| 2022-04-07 12:40:22.470078+05:30 | 60
```

4.2.5.3 Getting Project Reports

4.2.5.3.i Synopsis

amgr report project -n <project name> [-s <service unit> | -t <service unit type>] [-U] [-h] [-p <period>] [-S <start date>] [-E <end date>] [-l] [-o <output file>] [-r]

4.2.5.3.ii Description

Produces a project report.

By default, this prints information for all standard service units for the current period that is lowest in the hierarchy (shortest time division). You can refine your output by specifying service unit name and type.

By default, this prints the report to the screen in human-readable tables. You can print to a file, and you can print the report in raw (CSV) format.

4.2.5.3.iii Options

-n, --name <project name>

String. Name of project on which to get report.

-s, --serviceunit <service unit name>

Prints report for specified service unit.

If you do not specify a service unit, the report includes all service units of the specified type. If you do not specify type, it is *SU_STANDARD*.

-t, --sunit-type <service unit type>

Use this option to see dynamic service units. By default, this command prints only standard service units.

Cannot be used with `-l`, `-g`, or `-h` options.

Type can be one of *SU_STANDARD* or *SU_DYNAMIC*.

Default: *SU_STANDARD*

-U, --user-wise

Prints consumption, one line per consumer, per service unit. Output includes amount of credit, username, and period.

By default only standard service units are printed; to see dynamic service units, use the -t option.

-g, --global-view

Lists project details and a summary of all transactions with a detailed listing of each transaction in JSON format.

Cannot be used with `-t SU_DYNAMIC`.

-h, --stakeholder-info

Lists groups which have invested in the project and their invested amounts.

Cannot be used with `-t SU_DYNAMIC`.

-p, --period <period name>

Specifies report period.

Cannot be used with `-S` and `-E` options.

Default: current period that is lowest in hierarchy (shortest time division)

-S, --start-date <start date>

Reports activity starting at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-E, --end-date <end date>

Reports activity ending at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-l, --list-info

Lists all transactions that have happened for the specified project.

Use the `-l` option to get the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type, serviceunit, amount, balance, reconciled, period, am_mode, and comment, in columns.

Cannot be used with `-t SU_DYNAMIC`.

-o, --out-file <output filename>

Budgets prints the report in a human-readable form to the specified output file.

Without this option, Budgets prints the report to the screen.

-r, --raw-output

Print report in raw (CSV) form with no padding. This form is easier to parse but harder for a human to read.

4.2.5.3.iv Output Format

The report is printed as a columns, for name, start_date, end_date, opening_balance, total_credits, total_debits_reconciled, total_debits_authorized, and net_balance.

4.2.5.3.v Command Example for Postpaid Mode

Example 4-29: Project report in postpaid mode:

```
amgr report project -n project1
```

Command output:

```
-----------------------------------------------------------------------
name       | period   | serviceunit   | total_outstanding   | metadata
-----------------------------------------------------------------------
project1   | 2022     | cpu_hrs       | 1440.0              | {}
```

4.2.5.3.vi Command Example for Prepaid Mode

Example 4-30: Report for all standard service units and current lowest period:

```
amgr report project -n p1
```

Command output:

```
----------------------------------------------------------------------
name | period   | serviceunit | opening_balance | total_credits
----------------------------------------------------------------------
p1   | DEC-2018 | dollar1     | 0.0             | 3000000.0

-----------------------------------------------------------------------
| total_debits | total_debits_reconciled | total_debits_authorized
-----------------------------------------------------------------------
| 0.0          | 180.0                   | 0.0

--------------------------
| net_balance | metadata
--------------------------
| 2999820.0   | {}
```

Example 4-31: Report for service unit dollar1 for period 2018:

```
amgr report project -n p1 -s dollar1 -p 2018
```

Command output:

```
-----------------------------------------------------------------------
name | period | serviceunit | opening_balance | total_credits
-----------------------------------------------------------------------
p1   | 2018   | dollar1     | 0.0             | 5000000.0

-----------------------------------------------------------------------
| total_debits | total_debits_reconciled | total_debits_authorized
-----------------------------------------------------------------------
| 0.0          | 0.0                     | 0.0

-----------------
| net_balance
-----------------
| 5000000.0
```

To get raw output for the project:

```
amgr report project -n p1 -s dollar1 -p 2018 -r
```

Command output:

```
name,period,serviceunit,opening_balance,total_credits,total_debits,total_debits_reconciled,total
_debits_authorized,net_balance
p1,2018,dollar1,0.0,5000000.0,0.0,0.0,0.0,5000000.0
```

Example 4-32: Show individual transactions for the service unit dollar1 for the lowest period:

```
amgr report project -n p1 -s dollar1 -l
```

Note that this prints the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type, serviceunit, amount, balance, reconciled, period, am_mode, and comment, in columns.

Command output:

```
---------------------------------------------------------------------------
transaction_id    | transaction_date | transaction_time | entity
---------------------------------------------------------------------------
1544096706.291656 | 2018-12-06       | 17:15:06.290064  | manager
2042.cluster1     | 2018-12-06       | 17:15:23.760451  | job
2043.cluster1     | 2018-12-06       | 17:15:24.695462  | job
2044.cluster1     | 2018-12-06       | 17:16:24.058123  | job

---------------------------------------------------------------------------------
| transaction_type | transaction_user | type   | serviceunit | amount
---------------------------------------------------------------------------------
| grant            | Manager1          | credit | dollar1     | 3000000.0
| acquired         | amteller          | debit  | dollar1     | 60.0
| acquired         | amteller          | debit  | dollar1     | 60.0
| acquired         | amteller          | debit  | dollar1     | 60.0

-------------------------------------------------------------
| reconciled | period   | comment
-------------------------------------------------------------
| yes        | DEC-2018 | Deposit dollar1 to DEC period
| yes        | DEC-2018 |
| yes        | DEC-2018 |
| yes        | DEC-2018 |
```

Example 4-33: Print individual transactions for the service unit dollar1 for a date range:

```
amgr report project -n p1 -s dollar1 -l -S '2018-12-06' -E '2018-12-06'
```

Note that this prints the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type, serviceunit, amount, balance, reconciled, period, am_mode, and comment, in columns.

Command output:

```
-----------------------------------------------------------------------------
transaction_id    | transaction_date | transaction_time | entity
-----------------------------------------------------------------------------
1544096706.100772 | 2018-12-06       | 17:15:06.098180  | manager
1544096706.291656 | 2018-12-06       | 17:15:06.290064  | manager
2042.cluster1     | 2018-12-06       | 17:15:23.760451  | job
2043.cluster1     | 2018-12-06       | 17:15:24.695462  | job
2044.cluster1     | 2018-12-06       | 17:16:24.058123  | job

----------------------------------------------------------------------------------
| transaction_type | transaction_user | type   | serviceunit | amount
----------------------------------------------------------------------------------
| grant            | Manager1          | credit | dollar1     | 5000000.0
| grant            | Manager1          | credit | dollar1     | 3000000.0
| acquired         | amteller          | debit  | dollar1     | 60.0
| acquired         | amteller          | debit  | dollar1     | 60.0
| acquired         | amteller          | debit  | dollar1     | 60.0

-----------------------------------------------------------------
| reconciled | period   | comment
-----------------------------------------------------------------
| yes        | 2018     | Deposit dollar1 to parent period
| yes        | DEC-2018 | Deposit dollar1 to DEC period
| yes        | DEC-2018 |
| yes        | DEC-2018 |
| yes        | DEC-2018 |
```

4.2.5.3.vii Project Reports Showing Dynamic Service Units

Example 4-34: Project report in short format:

```
amgr report project -n p1 -t SU_DYNAMIC
--------------------------------------------------------------------------------------
name | serviceunit | period | limit |last_reported_time               | total_consumed
--------------------------------------------------------------------------------------
p1   | luster      | 2022   | 500.0 |2022-04-07 12:40:22.470078+05:30 | 80
p1   | scratch     | 2022   | 800.0 |2022-04-07 12:40:22.470078+05:30 | 100
```

Example 4-35: Project report in user-wise format:

```
amgr report project -n p1 -t SU_DYNAMIC -U


---------------------------------------------------------------------------------------
name | serviceunit | period | limit |last_reported_time | total_consumed | user_consumed
---------------------------------------------------------------------------------------
p1  | luster    | 2022   | 500.0 | 2022-04-07 12:40   |  80   | {"u1": 20.0, "u2": 10.0}
p1  | scratch   | 2022   | 800.0 | 2022-04-07 12:40   | 100   | {"u1": 40.0}
```

Example 4-36: Report project storage data:

```
amgr report project -n proj1 -t SU_DYNAMIC
```

Command output:

```
---------------------------------------------------------------------------------------
name  | serviceunit | period  | limit | last_reported_time               | total_consumed
---------------------------------------------------------------------------------------
proj1 | storage     | 2022.feb| 25.0  | 2022-02-11 18:58:28.270385+00:00 | 30.0
```

4.2.5.4 Getting Group Reports

4.2.5.4.i Synopsis

amgr report group -n <group name> [-h] [-A] [-S <start-date>] [-E <end-date>] [-l] [-o <output-file>] [-r] [-p <period>]

4.2.5.4.ii Description

Produces group report showing deposits and withdrawals.

By default, this prints information for all standard service units for the current period that is lowest in the hierarchy (shortest time division). You can refine your output by specifying service unit name.

By default, this prints the report to the screen in human-readable tables. You can print to a file, and you can print the file in raw (CSV) format.

4.2.5.4.iii Options

-n, --name <group name>

String. Name of the group on which to get report.

-h, --stakeholder-info

For each investor in the group, lists remaining balance for that investor, of what they invested.

-A, --allocated-view

Lists remaining credit allocated from this group to users and projects.

Shows for each project, how much this project has remaining of what the group invested.

Shows for each user, how much this user has remaining of what the group invested.

Cannot filter by project or user.

-S, --start-date <start date>

Reports activity starting at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-E, --end-date <end date>

Reports activity ending at this date. Report includes created and updated transactions.

Requires `-l` or `-g` option.

Cannot be used with `-p` option.

Format: YYYY-MM-DD HH:MM:SS

-l, --list-info

Lists all transactions that have happened for the specified group.

Use the `-l` option to get the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type amount and reconciled status in columns.

-o, --out-file <output filename>

Budgets prints the report in a human-readable form to the specified output file.

Without this option, Budgets prints the report to the screen.

-r, --raw-output

Print report in raw (CSV) form with no padding. This form is easier to parse but harder for a human to read.

-p, --period-name <period name>

Specifies report period.

Cannot be used with `-S` and `-E` options.

Default: current period that is lowest in hierarchy (shortest time division)

4.2.5.4.iv Group Report Examples

Example 4-37: Default group report:

```
amgr report group -n h1
```

Command output:

```
--------------------------------------------------------------------
name | serviceunit | opening_balance | total_credits | total_debits
--------------------------------------------------------------------
h1   | cpu_hrs     | 0.0             | 100000.0      | 100.0

-----------------------------------------------------------------
| total_allocated | total_accounts_released | net_balance
-----------------------------------------------------------------
| 600.0           | 70.0                    | 99370.0
```

Example 4-38: Long format group report:

```
amgr report group -n h1 -l
```

Command output:

```
-------------------------------------------------------------------------------------
transaction_id     | transaction_date | transaction_time | entity   | transaction_type
-------------------------------------------------------------------------------------
1570782001.2559264 | 2022-10-11       | 13:50:01.253271  | investor | grant
2470123401.5592132 | 2022-10-11       | 13:50:11.967594  | manager  | grant
6470127408.2212197 | 2022-10-11       | 13:50:31.161543  | manager  | grant

-----------------------------------------------------------------------------
transaction_user | type   | serviceunit | amount   | balance  | period | account
-----------------------------------------------------------------------------
root             | credit | cpu_hrs     | 500000.0 | 500000.0 | -      | group1
mgr1             | credit | cpu_hrs     | 1000.0   | 499996.0 | 2022   | Project1
mgr2             | credit | cpu_hrs     | 4.0      | 499992.0 | 2022   | user1
```

Example 4-39: Report in investor format:

```
amgr report group -n h1 -h
```

Command output:

```
----------------------------------------------------------
investor | serviceunit | balance
----------------------------------------------------------
root     | cpu_hrs     | 50000.0
rsv      | cpu_hrs     | 50000.0
```

Example 4-40: Report for primary group accounts:

```
amgr report user -n h1 -h
```

Command output:

```
----------------------------------------------------------------
group | period | serviceunit | balance
----------------------------------------------------------------
h1    | 2022   | cpu_hrs     | 1500.0
h2    | 2022   | cpu_hrs     | 1500.0
```

Example 4-41: Group report in investor format:

This report shows where managers have deposited the group's service units.

```
amgr report group -n h1 -A
```

Command output:

```
--------------------------------------------------------------
account | period | serviceunit | balance
--------------------------------------------------------------
root    | 2022   | cpu_hrs     | 498976.0
rsv     | 2022   | cpu_hrs     | 498975.0
```

4.2.5.5 Getting Job and Transaction Reports

4.2.5.5.i Synopsis

amgr report transaction [-i <job ID or transaction ID>] [-l] [-N <count>] [-o <output-file>] [-r]

4.2.5.5.ii Description

Produces a report for a job or transaction.

The default report for a job ID is the net cost for the job.

The default report for a transaction ID is a summary of that transaction.

By default, this prints the report to the screen in human-readable tables. You can print to a file, and you can print the report in raw (CSV) format.

4.2.5.5.iii Options

-i, --transaction-id <transaction ID>

String. ID of Job or transaction on which to get report.

-l, --list-info

Lists all transactions that have happened for the specified job or transaction ID.

Use the `-l` option to get the report in long format, containing the transaction_id, transaction_date, transaction_time, entity, transaction_type, transaction_user, type amount and reconciled status in columns.

-N, --non-reconcile <count>

For use without job or transaction ID. Show all non-reconciled jobs that have had *count* or more attempts to reconcile.

For example, `amgr report transaction -N 2` displays all non-reconciled jobs with count >=2.

Budgets attempts to reconcile a job three times; if this doesn't work, Budgets marks the job as non-reconciled.

-o, --out-file <output filename>

Prints the report in a human-readable form to the specified output file.

Without this option, Budgets prints the report to the screen.

-r, --raw-output

Print report in raw (CSV) form with no padding. This form is easier to parse but harder for a human to read.

4.2.5.5.iv Output Format

Output for the report lists columns for transaction_id, project, user_name, and reconciled_service_units.

If a job is not reconciled, it appears only when you use the `-l` option for a long format report.

Use the `-l` switch to get the report for the job or transaction in long format, which lists transaction_id, transaction_date, transaction_time, project, transaction_type, credit or debit, service units, and amounts.

4.2.5.5.v Examples

Example 4-42: Print job report:

```
amgr report transaction -i 2044.cluster1
```

Command output:

```
----------------------------------------------------------------------------------
transaction    | account | user     | reconciled_dollar1 | reconciled_dollar2
----------------------------------------------------------------------------------
2044.cluster1 | p1      | pbsuser | 60.0               | 21120.0
```

Example 4-43: Print long format report for job, showing individual operations:

```
amgr report transaction -i 2044.cluster1 -l
```

Command output:

```
-----------------------------------------------------------------------
transaction_id  | transaction_date | transaction_time | account
-----------------------------------------------------------------------
2044.cluster1   | 2018-12-06       | 17:17:46.655998  | p1
2044.cluster1   | 2018-12-06       | 17:16:24.058123  | p1
2044.cluster1   | 2018-12-06       | 17:16:24.058123  | p1

------------------------------------------------
| transaction_type | type  | service_unit
------------------------------------------------
| acquired         | debit | dollar2
| acquired         | debit | dollar2
| acquired         | debit | dollar1

-------------------------------------
| amount  | period   | comment
-------------------------------------
| 19920.0 | DEC-2018 | overrun:
| 1200.0  | DEC-2018 |
| 60.0    | DEC-2018 |
```

4.2.6 Applying Limits to Dynamic Service Units

amgr limit {user | project}

Apply limits on dynamic service units for projects or users.

4.2.6.1 Synopsis

amgr limit user -n <username> -p <period name> -s <service unit name> <limit value>

amgr limit project -n <project name> -p <period name> -s <service unit name> <limit value>

4.2.6.2 Description

You establish a quota for an external resource by applying a limit to the dynamic service unit representing that resource. Usage of the external resource is reported by an external application. Jobs that use this resource can start only while the job owner has not reached the quota for this period.

Standard service units don't need a limit.

A limit on a dynamic service unit applies to a particular period and for a particular project or user account.

You can apply limits to any period.

All active dynamic service units must have limits set for the current period for jobs to run.

4.2.6.3 Effect of Limits on the Period Hierarchy

- If a parent period has a limit set for a particular dynamic service unit, its children inherit that limit unless the limit is explicitly set by the group manager.
- If you apply a limit to a period, that value overrides any inherited value.
- The default limit is zero unless the period in question inherits a parent's value.

4.2.6.3.i Rules for LImits on Dynamic Service Units

If a dynamic service unit has no limit set on it, the limit is zero. If there are active dynamic service units, all jobs are checked against those quotas, and a zero quota will stop any job from running. Make sure that you don't unintentionally stop non-target users or projects from running jobs:

- Make sure you set the quota for all users and projects
- When you specify the period, make sure you either:
    - Set the desired quota at the top level of the period hierarchy
    - Set a very high quota at the top level of the period hierarchy, and a more restrictive quota for the period you need to control

4.2.6.4 Required Privilege

You must be a manager associated with the group funding the user or project to apply limits to a dynamic service unit for that user or project.

4.2.6.5 Options

-p, --period <period name>

String. Period to which the limit is to be applied.

-s, --serviceunit <service unit name>

String. Service unit to limit.

<limit value>

Float. Budgets reads this as being in the same units as the dynamic service unit.

Default: If a limit is unset, it is zero.

4.2.6.6 Examples

Example 4-44: Set a limit for the project MyProject, for the period named MyQuarter, on the service unit luster, to be 100:

```
amgr limit project -n MyProject -p MyQuarter -s luster 100
```

Example 4-45: Set a quota on storage of 12.0 for user user1 for the period of 2022:

```
amgr limit user -n user1 -s storage 12.0 -p 2022
```

Example 4-46: Set a quota of 25 for storage for the project project1 for the period of 2022:

```
amgr limit project -n project1 -s storage 25.0 -p 2022
```

4.2.7 Syncing Formula File to PBS Complex

4.2.7.1 Synopsis

amgr sync formula -c <cluster>

4.2.7.2 Description

Pulls formula file from the cluster data structure representing the specified PBS complex, pushes it to the specified complex, and imports it into the am_hook and am_hook_periodic hooks at the specified complex.

4.2.7.3 Required Privilege

You must be *admin* to run this command.

4.2.7.4 Options

-c, --cluster <target PBS complex>
String. Name of PBS complex where formula file is to be updated.

4.3 Transaction and Account Checking Commands

The Budgets service units commands and their respective subcommands are listed here:

Table 4-3: Budgets Service Units Commands

Function	Command	Element Subcommands	Required Privilege	Link
Depositing service units	`amgr deposit`	`user, project, group`	*investor* for deposit to group Group *manager* for deposit to user or project account	Depositing Service Units
Checking balance of service units	`amgr check-balance`	`user, project, group`	*user* can check own balance	Checking Service Unit Balance
Withdrawing service units	`amgr withdraw`	`user, project, group`	*investor* to withdraw from group Group *manager* to withdraw from user or project account	Withdrawing Service Units
Transferring service units	`amgr transfer`	`user, project, group`	*admin*	Transferring Service Units
Prechecking service unit balance	`amgr precheck`	`user, project`	*user* can precheck own balance *admin* or *teller* to check other balances	Prechecking Service Unit Balance
Acquiring service units	`amgr acquire`	`user, project`	*admin* or *teller*	Acquiring Service Units
Reconciling service units	`amgr reconcile`	`user, project`	*admin* or *teller*	Reconciling Service Units
Refunding service units	`amgr refund`	`transaction`	*admin*	Refunding Service Units

4.3.1 Depositing Service Units

amgr deposit {user | project | group}

4.3.1.1 Deposit Service Units to User

4.3.1.1.i Synopsis

amgr deposit user -n <username> -s <service unit name> <service unit amount> -p <period> -h <group> [-C <comment>]

4.3.1.1.ii Description

Deposits service units to a user account.

4.3.1.1.iii Required Privilege

You must be *manager* to run this command.

4.3.1.1.iv Options

-n, --name <username>

String. User account to receive service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Specifies the service unit to be deposited and the quantity to deposit.

-p, --period <period>

String. Period during which service units are available.

-h, --group <group name>

String. Name of the group from which service units are to be allocated.

-C, --comment <comment>

String. Comment or reason for the deposit. Optional.

Default: no comment

4.3.1.2 Depositing Service Units to Project

4.3.1.2.i Synopsis

amgr deposit project -n <project name> -s <service unit name> <service unit amount> -p <period> -h <group> [-C <comment>]

4.3.1.2.ii Description

Deposits service units to a project.

4.3.1.2.iii Required Privilege

You must be *manager* to run this command.

4.3.1.2.iv Options

-n, --name <project name>

String. Name of project to receive service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Specifies the service unit to be deposited and the quantity to deposit.

-p, --period <period>

String. Period during which service units are available.

-h, --group <group name>

String. Name of the group from which service units are to be allocated.

-C, --comment <comment>

String. Comment or reason for the deposit. Optional.

Default: no comment

4.3.1.3 Depositing Service Units to Group

4.3.1.3.i Synopsis

amgr deposit group -n <group name> -s <service unit name> <service unit amount> [-C <comment>]

4.3.1.3.ii Description

Deposits investor service units to a group. This command is run by the investor; the funds invested come from the investor running the command.

4.3.1.3.iii Required Privilege

This command must be run by an *investor*.

4.3.1.3.iv Options

-n, --name <group name>

String. Name of group to receive service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Specifies the service unit to be deposited and the quantity to deposit.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-C, --comment <comment>

String. Comment or reason for the deposit. Optional.

Default: no comment

4.3.2 Checking Service Unit Balance

amgr checkbalance {user | project | group}

4.3.2.1 Required Privilege

A *user* can check their own service unit balance.

A member of a project can check the service unit balance for that project.

An *admin, manager,* or *investor* can check the service unit balance for all groups and projects.

4.3.2.2 Output Format

- For `amgr checkbalance {user | project}`, output is a list of standard service units in name-value pairs and dynamic service units in name-limit-value triplets, in the order in which the service units were created:

```
{"<standard unit 1>": <amount available>,
 "<standard unit 2>": <amount available>,
 "<dynamic unit 1>": {"used": <amount used>, "limit": <limit value>},
 "<dynamic unit 2>": {"used": <amount used>, "limit": <limit value>} }
```

- For `amgr checkbalance group`, output is a list of standard service units in name-value pairs, in the order in which the service units were created:

```
{"<standard unit 1>": <amount available>,
 "<standard unit 2>": <amount available>,}
```

4.3.2.3 Command Example

Example 4-47: For output from `amgr checkbalance {user | project}`, dynamic service unit luster, and standard service units cpu_hrs and gpu_hrs:

```
{"luster": {"used": 0.0, "limit": 300}, "cpu_hrs":0.0, "gpu_hrs":0.0}
```

Example 4-48: For output from `amgr checkbalance group`, standard service units cpu_hrs and gpu_hrs:

```
{"cpu_hrs":0.0, "gpu_hrs":0.0}
```

4.3.2.4 Checking Service Unit Balance for User

4.3.2.4.i Synopsis

amgr checkbalance user -n <username> [-p <period>]

4.3.2.4.ii Description

Check available service unit balance for a user account.

4.3.2.4.iii Options

-n, --name <username>

String. User account to check for available service units.

-p, --period <period>

String. Period during which service units are required.

Default: period is the current period that is lowest in the hierarchy, meaning shortest time span.

4.3.2.5 Checking Service Unit Balance for Project

4.3.2.5.i Synopsis

amgr checkbalance project -n <project name> -p <period>

4.3.2.5.ii Description

Checks available service unit balance for a project.

4.3.2.5.iii Options

-n, --name <project name>

String. Name of project to check for available service units.

-p, --period <period>

String. Period during which service units are required.

4.3.2.6 Checking Service Unit Balance for Group

4.3.2.6.i Synopsis

amgr checkbalance group -n <group name>

4.3.2.6.ii Description

Checks available service unit balance for a group.

4.3.2.6.iii Options

-n, --name <group name>

String. Name of group to check for available service units.

4.3.3 Withdrawing Service Units

amgr withdraw {user | project | group}

4.3.3.1 Withdrawing Service Units from User

4.3.3.1.i Synopsis

amgr withdraw user -n <username> -s <service unit type> <service unit amount> <-p <period> -h <group> [-C <comment>]

4.3.3.1.ii Description

Withdraw service units from a user account. You can withdraw only up to the amount you have deposited.

4.3.3.1.iii Required Privilege

You must be *manager* to run this command.

4.3.3.1.iv Options

-n, --name <username>

String. User account from which to withdraw service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit to be withdrawn and the quantity to withdraw.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-p, --period <period>

String. Period from which to withdraw service units.

-h, --group <group name>

String. Group to receive withdrawn service units.

-C, --comment <comment>

String. Comment or reason for the withdrawal. Optional.

Default: no comment

4.3.3.2 Withdrawing Service Units from Project

4.3.3.2.i Synopsis

amgr withdraw project -n <project name> -s <service unit type> <service unit amount> <-p <period> -h <group> [-C <comment>]

4.3.3.2.ii Description

Withdraws service units from a project.

A manager can withdraw only up to the amount they deposited.

4.3.3.2.iii Required Privilege

You must be *manager* to run this command.

4.3.3.2.iv Options

-n, --name <project name>

String. Name of project from which to withdraw service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit to be withdrawn and the quantity to withdraw.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-p, --period <period>

String. Period from which to withdraw service units.

-h, --group <group name>

String. Group to receive withdrawn service units.

-C, --comment <comment>

String. Comment or reason for the withdrawal. Optional.

Default: no comment

4.3.3.3 Withdrawing Service Units from Group

4.3.3.3.i Synopsis

amgr withdraw group -n <group name> -s <service unit type> <service unit amount> [-C <comment>]

4.3.3.3.ii Description

Withdraw service units from a group. The investor can remove only the amount of service units that the investor deposited.

4.3.3.3.iii Required Privilege

You must be *investor* to run this command.

4.3.3.3.iv Options

-n, --name <group name>

String. Group from which to withdraw service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit to be withdrawn and the quantity to withdraw.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-C, --comment <comment>

String. Comment or reason for the withdrawal. Optional.

Default: no comment

4.3.4 Prechecking Service Unit Balance

amgr precheck {user | project | jobs}

4.3.4.1 Prechecking a User or Project

4.3.4.1.i Synopsis

amgr precheck user -n <username> -c <cluster> [-D <transaction date>] -d <duration> -s <service unit name> <service unit amount> -u <username> -f <formula file>

amgr precheck project -n <project name> -c <cluster> [-D <transaction date>] -d <duration> -s <service unit name> <service unit amount> -u <username> -f <formula file>

4.3.4.1.ii Description

This command is intended to be used internally for prechecking jobs when they are submitted, and you do not need to run it in the normal course of events. It is used to check whether the resources needed to run a particular job are available. Checks whether the project or user has sufficient funds and the specified complex is available to run a job at a specified date for a specific amount of service units.

If AM_BALANCE_PRECHECK is *True* and the check fails, the job is not queued. If AM_BALANCE_PRECHECK is *False* and the check fails, the job is queued.

4.3.4.1.iii Required Privilege

A *user* can precheck their own account.

You must be *admin* or or *teller* to run this command for any other account.

4.3.4.1.iv Options

-n, --name <project name>

String. Name of the project or user to check for available service units.

-c, --cluster <PBS server>

String. PBS complex to check for availability.

-D, --transaction-date <transaction date>

Date. Transaction date and time.

Format: YYYY-MM-DD HH:MM:SS

-d, --duration <duration>

Integer seconds.

Job duration, in seconds.

-s, --serviceunits <service unit name> <service unit value>

String and float. Name of service unit and required amount.

-u, --user <username>

String. User account for whom to precheck service units.

-f, --formula <formula filename>

String. Formula file for cluster.

4.3.4.1.v Output

If the job owner has sufficient credit to run the specified job, there is no output. If the job owner does not have enough credit to run the job, this command prints a message to the job submitter's screen indicating the credit shortfall.

4.3.4.2 Prechecking Jobs

4.3.4.2.i Synopsis

amgr precheck jobs [-v <job string> | -J <job file>] -f <formula file>

4.3.4.2.ii Description

Allows the job submitter to make sure that their credit is sufficient to run jobs.

To use this command on a job, you need to know how much of each service unit is required for that job, so you may need to first get a quote for the job. For job quotes, see "Getting Job Cost Estimate from Budgets", on page 197 of the PBS Professional User's Guide.

The command prints a JSON string of key-value pairs. Each key is a job ID and the corresponding value is *True* or *False*. *True* indicates that the user has sufficient credit to run that job.

You give the command a list of jobs, and the command considers each job in turn. If a job could run, the credit required for that job is not included in the test for the next job. For example, you have 10 service units and you are testing 4 jobs needing 20, 4, 4, and 4 units respectively. In this case, the first job cannot run, the second job is tested against 10 service units and can run, the third job is tested against 6 service units and can run, and the fourth job is tested against 2 service units and cannot run.

The input job ID does not need to be the identifier of an existing job; it is used only to identify whether that job could run.

4.3.4.2.iii Required Privilege

A *user* can precheck their own jobs.

You must be *admin* or or *teller* to run this command for any other account.

4.3.4.2.iv Options

-v, --value <job string>

String. JSON string listing job(s) to be prechecked.

Format:

```
'{
    "<job ID>":{
        "account":<account name>,
        "cluster":<cluster name>,
        "user":<username>,
        "transaction_date":<date>,
        "duration":<duration>,
        "serviceunits":{
            "<service unit1 name>":"<value>","<service unit2 name>":"<value>",...
        }
    }, ...
}'
```

Where:

<job ID>
String used to identify job. Can be arbitrary or can be existing job.

account
String. Project or user account name.

cluster
String. Cluster name.

user
: String. Username of job submitter.

transaction_date
: Datetime. Transaction date. Format:

  'YYYY-MM-DD HH-MM-SS'

duration
: Duration. Time required for job to run.

  Format: integer seconds

serviceunits
: String. List of key-value pairs. Each key is the name of a service unit, and each value is the amount of that service unit required by the job.

Cannot be used with `-J` option.

-J, --json-file <JSON file>

Path to JSON input file listing job(s) to be prechecked. Path can be absolute or relative to directory where command is run.

Format: same as JSON *job string* argument to `-v` option.

Cannot be used with `-v` option.

-f, --formula <formula filename>

String. Path to formula file for cluster. Required. Path can be absolute or relative to directory where command is run.

4.3.4.2.v Output

The command prints a list of key-value pairs in JSON format. Each key is a job ID, and each value is *True* or *False*, where *True* indicates that the user has sufficient credit to run that job.

4.3.4.2.vi Examples

Example 4-49: Check a job using a JSON input string:

```
amgr precheck jobs -v '{"0.myserver":{"account":"project1","cluster":"myser-
   ver","user":"myuser","transaction_date":"2021-11-10 14:27:30","duration":100,"service-
   units":{"cpu_hrs":200.0,"dollar":5.0}}' -f /opt/am/hooks/pbs/my_formula.json
```

Response:

```
{"o.myserver":True}
```

Example 4-50: Check two jobs using a JSON input file:

```
amgr precheck jobs -J /tmp/precheck_data.json -f /opt/am/hooks/pbs/my_formula.json
```

Response:

```
{"o.myserver":True,"1.myserver":False}
```

Where:

```
cat /tmp/precheck_data.json
{
    "0.myserver":{
        "account":"project1",
        "cluster":"myserver",
        "user":"myuser",
        "transaction_date":"2021-11-10 14:27:30",
        "duration":100,
        "serviceunits":{
            "cpu_hrs":200.0,
            "dollar":5.0
        }
    }
    "1.myserver":{
        "account":"myuser",
        "cluster":"myserver",
        "user":"myuser",
        "transaction_date":"2021-11-10 14:27:40",
        "duration":50,
        "serviceunits":{
            "cpu_hrs":100.0,
            "dollar":2.0
        }
    }
}
```

4.3.5 Acquiring Service Units

amgr acquire {user | project}

4.3.5.1 Description

This command is intended to be used for debugging. You do not need to run it in the normal course of events. The Budgets hook performs this operation for normal job processing.

Fetches the service units that are required to run a job from the user or project.

The teller uses this command to acquire service units for a user by taking them from a project or user account.

Run the checkbalance command before running a job to make sure there are enough service units available.

4.3.5.2 Required Privilege

You must be an *admin* or *teller* to run this command.

4.3.5.3 Acquiring Service Units for User

4.3.5.3.i Synopsis

amgr acquire user -n <username> -c <cluster> -s <service unit name> <service unit amount> [-D <transaction date>] -i <job ID> -d <duration> -u <username> [-C <comment>] -R <run count> -f <formula file>

4.3.5.3.ii Options

-n, --name <username>
: String. User account from which to fetch service units.

-c, --cluster <PBS server>
: String. Name of cluster.

-s, --serviceunits <service unit name> <service unit amount>
: String and float. Name of service unit and the quantity required.
: Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-D, --transaction-date <date>
: Date. Transaction date and time.
: Format: YYYY-MM-DD HH:MM:SS

-i, --transaction-id <job ID>
: Full PBS job ID.

-d, --duration <duration>
: Duration. Job duration, in seconds.

-u, --user <username>
: String. User account for whom to acquire service units.

-C, --comment <comment>
: String. Comment or reason for the acquisition. Optional.
: Default: no comment

-R, --run-count <run count>
: Integer. Run count for the job.

-f, --formula <formula filename>
: String. Formula file for cluster.

4.3.5.4 Acquiring Service Units for Project

4.3.5.4.i Synopsis

amgr acquire project -n <project name> -c <cluster> -s <service unit name> <service unit amount> -D <transaction date> -i <job ID > -d <duration> -u <username> [-C <comment>] -R <run count> -f <formula file>

4.3.5.4.ii Options

-n, --name <project name>
: String. Project from which to fetch service units.

-c, --cluster <PBS server>
: String. Name of cluster.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit and the quantity required.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-D, --transaction-date <date>

Date. Transaction date and time.

Format: YYYY-MM-DD HH:MM:SS

-i, --transaction-id <job ID>

Full PBS job ID.

-d, --duration <duration>

Duration. Job duration, in seconds.

-u, --user <username>

String. User account for whom to acquire service units.

-C, --comment <comment>

String. Comment or reason for the acquisition. Optional.

Default: no comment

-R, --run-count <run count>

Integer. Run count for the job.

-f, --formula <formula filename>

String. Formula file for cluster.

4.3.6 Reconciling Service Units

amgr reconcile {user | project}

Reconciles service units charged and consumed for jobs. Removes consumed service units from escrow, and returns unused service units to the account, or if the job consumed more than requested it debits more from the job owner's account.

This command is mostly intended to be used for cleanup and debugging. You do not need to run it in the normal course of events. The Budgets hook performs all reconcile operations during normal job processing.

4.3.6.1 Required Privilege

You must be root and *admin* or *teller* to run this command. You must run this command at the PBS server host.

4.3.6.2 Reconciling Service Units for User

4.3.6.2.i Synopsis

amgr reconcile user -n <username> -c <cluster> -d <duration> -f <formula file> -i <job ID> -s <service unit name> <service unit amount> -u <username> [-D <transaction date>] [-C <comment>]

4.3.6.2.ii Description

Reconciles service units for job charged to a user account.

4.3.6.2.iii Options

-n, --name <username>

String. User for whom to reconcile service units.

-c, --cluster <PBS server>

String. Name of cluster.

-d, --duration <duration>

Integer. Job duration, in seconds.

-f, --formula <formula filename>

String. Formula file for cluster.

-i, --transaction-id <job ID>

Job ID.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit and the quantity actually consumed.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-u, --user <username>

String. Job submitter. User account for which service units are to be reconciled. Same as argument to `-n` option.

-D, --transaction-date <date>

Date. Transaction date and time.

Format: YYYY-MM-DD HH:MM:SS

-C, --comment <comment>

Comment or reason for the reconciliation. Optional.

Default: no comment

4.3.6.3 Reconciling Service Units for Project

4.3.6.3.i Synopsis

amgr reconcile project -n <project name> -c <cluster> -d <duration> -f <formula file> -i <job ID> -s <service unit name> <service unit amount> -u <username> [-D <transaction date>] [-C <comment>]

4.3.6.3.ii Description

Reconciles service units for job charged to a project account.

4.3.6.3.iii Options

-n, --name <project name>

String. Project for which to reconcile service units.

-c, --cluster <PBS server>

String. Name of cluster.

-d, --duration <duration>

Integer. Job duration, in seconds.

-f, --formula <formula filename>

String. Formula file for cluster.

-i, --transaction-id <job ID>

Job ID.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Name of service unit and the quantity actually consumed.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-u, --user <username>

String. Job submitter. User account for which service units are to be reconciled.

-D, --transaction-date <date>

Date. Transaction date and time.

Format: YYYY-MM-DD HH:MM:SS

-C, --comment <comment>

Comment or reason for the reconciliation. Optional.

Default: no comment

4.3.7 Refunding Service Units

amgr refund transaction

4.3.7.1 Description

Refunds some or all of the funds charged for a job to the user or project that funded the job.

Budgets knows who paid for the job, so you do not have to specify where the refund goes.

You can choose to provide a refund for situations such as when a job fails or runs multiple times, for reasons which cannot be attributed to the job owner. You can provide refunds to active and inactive projects and users. You can provide multiple refunds for the same job, but the total refund cannot exceed the amount consumed by the job.

The refund amount is calculated by multiplying the net job cost by the specified refund percentage by the total consumed amount. Total consumed amount is the sum of all transaction amounts of all transactions for a job.

Each group is refunded according to the percentage investment by that group. You can override this using the `-h` option.

4.3.7.2 Required Privilege

You must be *admin* to provide a refund.

4.3.7.3 Synopsis

amgr refund transaction -i <job ID> -r <refund percentage> -p <period> [-h <group>] [-C <comment>]

4.3.7.4 Options

-i, --transaction-id <job ID>

Job ID. Job for which to give a refund.

-r, --refund-percentage <refund percentage>

Integer between 1 and 100. Percentage of current net charge to refund.

-p, --period <period>

String. Period to refund.

Default: current period

-h, --group <group name>

String. Name of group which receives the refund. Overrides normal policy of refunding according to the percentage investment by each group. For use when you get an error message saying that Budgets cannot associate the refund to a group.

-C, --comment <comment>

String. Comment or reason for refund. Optional.

Default: no comment

4.3.8 Transferring Service Units

amgr transfer {user | project | group }

4.3.8.1 Description

Transfers service units from specified project, user, or group, to specified project, user, or group. You can transfer from any of these to any of these.

To transfer between investors, use `amgr transfer group`.

When an investor is unlinked from a group or a group is unlinked from a project, some service units may remain unclaimed and become unusable. The administrator can take ownership of these unclaimed amounts and transfer it back into the budget pool to make it usable again.

A transfer has its own transaction ID.

4.3.8.2 Required Privilege

You must be *admin* to run this command.

4.3.8.3 Transferring Service Units for User

4.3.8.3.i Synopsis

amgr transfer user -n <divesting username> -s <service unit name> <service unit amount> -p <period> -F <divesting group> -T <receiving group> [-U <receiving user>] [-C <comment>]

4.3.8.3.ii Description

Transfer service units from one user to another user, within a group or between groups.

4.3.8.3.iii Options

-n, --name <divesting username>

String. User account from which to take service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Service unit name and amount to transfer.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-p, --period <period>

String. Period from which to take service units.

-F, --from-group <divesting group>

String. Group from which to take service units.

-T, --to-group <receiving group>

String. Group to receive service units.

-U, --dest-user <receiving user>

String. User account to receive service units. Do not use when transferring within same user account.

-C, --comment <comment>

String. Comment or reason for the transfer. Optional.

Default: no comment

4.3.8.4 Transferring Service Units for Project

4.3.8.4.i Synopsis

amgr transfer project -n <divesting project> -s <service unit name> <service unit amount> -p <period> -F <divesting group> -T <receiving group> [-P <receiving project>] [-C <comment>]

4.3.8.4.ii Description

Transfers service units from one project to another, within a group or between groups.

4.3.8.4.iii Options

-n, --name <divesting project>

String. Project from which to take service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Service unit name and amount to transfer.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-p, --period <period>

String. Period from which to take service units.

-F, --from-group <divesting group>

String. Group from which to take service units.

-T, --to-group <receiving group>

String. Group to receive service units.

-P, --dest-project <receiving project>

String. Project to receive service units. Do not use when transferring within the same project.

-C, --comment <comment>

String. Comment or reason for the transfer. Optional.

Default: no comment

4.3.8.5 Transferring Service Units for Investors and Group

4.3.8.5.i Synopsis

amgr transfer group -n <divesting group> -s <service unit name> <service unit amount> -F <divesting investor> -T <receiving investor> -G <receiving group> [-C <comment>]

4.3.8.5.ii Description

Transfers service units from one investor to another investor, within a group or between groups.

4.3.8.5.iii Options

-n, --name <divesting group>

String. Group from which to take service units.

-s, --serviceunits <service unit name> <service unit amount>

String and float. Service unit and amount to transfer.

Examples: `-s cpu_hrs 100` or `-s cpu_hrs 100.0`

-F, --from-investor <divesting investor>

String. Investor from which to take service units.

-T, --to-investor <receiving investor>

String. Investor to receive service units.

-G, --dest-group <receiving group>

String. Group to receive service units. Do not use when transferring within group.

-C, --comment <comment>

String. Comment or reason for the transfer. Optional.

Default: no comment

5
Basic Install and Configure

5.1 Basic Install and Configure Instructions

5.1.1 Assumptions

- You install Budgets and AMS on the PBS server host
- You install Budgets in the default locations
- You use the default billing formula file

5.1.2 Installation

1. Log in as root.
2. Install utilities and `docker`:
   - For CentOS or RedHat:

     Log in as root to the service node (the machine where the AMS module is to be installed).

     ```
     yum install -y yum-utils
     yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
     yum install docker-ce docker-ce-cli containerd.io
     systemctl enable docker
     systemctl start docker
     yum install python3 python3-pip
     yum install openssl
     ```
   - For SLES12 or 15:

     Log in to the machine where Budgets is to be installed.

     For SLES 12:

     ```
         sudo SUSEConnect -p sle-module-containers/12/x86_64 -r ''
     ```
     For SLES 15:

     ```
         sudo SUSEConnect -p sle-module-containers/15.1/x86_64 -r ''
     sudo zypper install docker
     sudo systemctl enable docker.service
     sudo systemctl start docker.service
     ```
     Configure the firewall to allow forwarding of Docker traffic to the external network:

     ```
     Set FW_ROUTE="yes" in /etc/sysconfig/SuSEfirewall2
     zypper install python3-pip
     ```
   - For Ubuntu:

     Log in to the machine where Budgets is to be installed.

     ```
     sudo apt-get update
     sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-proper-
         ties-common
     curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
     sudo apt-key fingerprint 0EBFCD88
     ```
     The key should match the second line in the output; validate the last 8 characters. Example of second line:

     9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

     ```
     sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
         $(lsb_release -cs) stable"
     sudo apt-get update
     sudo apt-get install docker-ce docker-ce-cli containerd.io
     sudo apt-get install python3-pip
     ```
     This can install a number of required dependencies, and may take a few minutes.

     ```
     sudo apt-get install openssl
     sudo systemctl enable docker.service
     sudo systemctl start docker.service
     ```

3. On the Budgets server host, download the file containing both the Budgets server and the AMS module. The file-name has the following format:

   PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

   For example:

   PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.g

4. Untar the file:

tar xvfz PBSPro-budget-server_<release number>-<OS>_x86_64.tar.gz

For example:

```
tar xvfz PBSPro-budget-server_2022.1.0-CentOS7_x86_64.tar.gz
```

This creates the zip files for the Budgets server and the AMS module:

```
budget-manager-server-<OS>-<version>.tar.gz
ams-installer.zip
```

5. Copy `ams_installer.zip` over to the service node
6. On the service node, install AMS:

```
unzip ams-installer.zip
cd ams-installation
python3 -m pip install --upgrade --ignore-installed pbsworks-packager/
/usr/local/bin/pkgr
```
(Please stay in the AMS installer directory for this step)

7. Answer the questions in the dialogue:
   - Choose option: 0
   - Hit Enter until license agreement complete and then answer Yes to accept.
   - Select Enter to continue
   - Choose Option 1 (Provide server hostname/IP address)

     <hostname>

     <IP address>
   - Choose Option: 0 (Skips providing any more machine details)
   - Install Location: <hostname>
   - Authentication Server: <hostname>
   - Authentication Port: Accept Default (If alternative to default 22 for sshd is used then provide alternative port ID)
   - Provide *admin* username: pbsadmin
   - Install Path: Accept default
8. On the service node, edit `/etc/ssh/sshd_config` and add the following lines:

```
Match Address 10.5.0.0/24
PasswordAuthentication yes
```

9. On the service node, make `sshd` reread its configuration file, and restart it:

```
systemctl daemon-reload
systemctl restart sshd
```

10. On the Budgets server host, create certificates:

```
cd /home/pbsadmin/
mkdir budget_certificates
export AM_DBUSER=pbsdata
openssl req -new -x509 -days 3650 -nodes -out budget_certificates/ca.crt -keyout
    budget_certificates/ca.key -subj "/CN=root-ca"
openssl req -new -nodes -out server.csr -keyout budget_certificates/server.key -subj "/CN=local-
    host"
openssl x509 -req -in server.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
    budget_certificates/ca.key -CAcreateserial -out budget_certificates/server.crt
```

```
openssl req -new -nodes -out client.csr -keyout budget_certificates/client.key -subj
    "/CN=${AM_DBUSER}"
openssl x509 -req -in client.csr -days 3650 -CA budget_certificates/ca.crt -CAkey
    budget_certificates/ca.key -CAcreateserial -out budget_certificates/client.crt
rm -f server.csr client.csr
chmod og-rwx budget_certificates/*
```

11. Change to new directory called "am" created by untarring package earlier:

```
cd am/
```

12. Run the Budgets installer, and choose the username you want for the Budgets administrator. In our example we use *pbsadmin* for the administrator username:

```
./install -t server -u pbsadmin -c /home/pbsadmin/budget_certificates
```

5.1.3 Configuration

1. On the Budgets server host and any client hosts, edit /etc/am.conf, and set the parameters appropriately. See section 2.6.7.1, “Budgets Configuration Parameters”, on page 44
2. Set PATH for all users:

```
export PATH=/opt/am/python/bin:$PATH
```

3. On the Budgets/PBS server host, edit /etc/sudoers, and make it look like this:

```
Cmnd_Alias AM_SERVER_CMD = $AM _EXEC/python/bin/python3 $AM_EXEC/hooks/pbs/pbs_set_formula.py*
Defaults!AM_SERVER_CMD !requiretty
Cmnd_Alias AM_CLIENT_CMD = $AM_EXEC/python/bin/amgr sshlogin
Cmnd_Alias BUDGETS_IMPORTS = $PBS_EXEC/bin/qmgr -c i h am_hook application/x-config default
    .am/tmp*, $PBS_EXEC/bin/qmgr -c i h am_hook_periodic application/x-config default .am/tmp*,
    $PBS_EXEC/bin/qmgr -c export hook am_hook application/x-config default
pbsadmin ALL=(root) NOPASSWD: BUDGETS_IMPORTS
Defaults!AM_CLIENT_CMD !requiretty
Defaults!BUDGETS_IMPORTS !requiretty
```

4. Create resources:

```
qmgr -c "c r am_cloud_enabled type=boolean"
qmgr -c "c r am_job_amount type=string"
qmgr -c "c r am_job_cache type=string,flag=m"
qmgr -c "c r am_job_quote type=boolean"
qmgr -c "c r am_finished_job type=string"
qmgr -c "c r am_node_cache type=string"
qmgr -c "set server resources_available.am_finished_job=NA"
```

5. Define your billing periods.
6. Set up passwordless ssh for both pbsadmin and amteller. Do these steps once for each one:
   a. Log in to the PBS server as the username who needs passwordless ssh
   b. Check for an existing SSH key pair:

      ```
      ls -al ~/.ssh/id_*.pub
      ```

      If you find existing keys, you can use those or back up the old keys and generate a new one.

      To generate a new SSH key pair:

      ```
      ssh-keygen
      ```

   c. Copy the contents of id_rsa.pub
   d. Log in to the Budgets server as the username who needs passwordless ssh
   e. Check for the .ssh directory. If it does not exist, create it:

      ```
      mkdir -p .ssh
      cd .ssh/
      ```

   f. Create the authorized_keys file in the .ssh directory:

      ```
      vi authorized_keys
      ```

   g. Paste the contents of id_rsa.pub that you copied from the PBS server and save the file.
   h. Change the permission of authorized_keys to *600*:

```
chmod 600 authorized_keys
```

7. Log into the PBS server host
8. Create and configure the am_hook and am_hook_periodic hooks:
   - For prepaid mode:

```
qmgr -c "c h am_hook"
qmgr -c "s h am_hook event='queuejob,runjob,modifyjob,movejob'"
qmgr -c "s h am_hook order=1000"
qmgr -c "c h am_hook_periodic"
qmgr -c "s h am_hook_periodic event=periodic"
qmgr -c "s h am_hook_periodic freq=120"
qmgr -c "i h am_hook application/x-python default  /opt/am/hooks/pbs/am_hook.py"
qmgr -c "i h am_hook_periodic application/x-python default  /opt/am/hooks/pbs/am_hook.py"
qmgr -c "i h am_hook application/x-config default /opt/am/hooks/pbs/<formula file>.json"
qmgr -c "i h am_hook_periodic application/x-config default /opt/am/hooks/pbs/<formula
    file>.json"
qmgr -c "s h am_hook enabled=true"
qmgr -c "s h am_hook_periodic enabled=true"
qmgr -c "s h am_hook alarm=90"
```

   - For postpaid mode:

```
qmgr -c "c h am_hook"
qmgr -c "s h am_hook event='queuejob,modifyjob,movejob'"
qmgr -c "s h am_hook order=1000"
qmgr -c "c h am_hook_periodic"
qmgr -c "s h am_hook_periodic event=periodic"
qmgr -c "s h am_hook_periodic freq=120"
qmgr -c "i h am_hook application/x-python default  /opt/am/hooks/pbs/am_hook.py"
qmgr -c "i h am_hook_periodic application/x-python default  /opt/am/hooks/pbs/am_hook.py"
qmgr -c "i h am_hook application/x-config default /opt/am/hooks/pbs/<formula file>.json"
qmgr -c "i h am_hook_periodic application/x-config default /opt/am/hooks/pbs/<formula
    file>.json"
qmgr -c "s h am_hook enabled=true"
qmgr -c "s h am_hook_periodic enabled=true"
qmgr -c "s h am_hook alarm=90"
```

9. Set configuration parameters:

On the Budgets server host, make sure all of the configuration parameters in section 2.6.7.1, "Budgets Configuration Parameters", on page 44 are set correctly. Especially make sure that AM_MODE is set to the mode you want, because to change modes you need to restart Budgets. In addition, make sure that AM_AUTH_ENDPOINT and AM_LICENSE_ENDPOINT are set correctly.

10. Enable and start Budgets:
    - Enable and start Budgets:
      ```
      systemctl enable pbs_budget
      systemctl start pbs_budget
      ```
    - Check the status of Budgets:
      ```
      systemctl status pbs_budget
      ```
11. Validate Budgets:
    a. Log in as pbsadmin
    b. Test authentication:
       ```
       amgr login
       ```
    c. List users (pbsadmin and amteller):
       ```
       amgr ls user -l
       ```
12. At the Budgets server host, add a cluster to represent the PBS complex with its billing model:
    ```
    amgr add cluster <PBS server> -f am_hook.json
    ```

6
Using Budgets

6.1 Managing Credit with Budgets

Administrators must log into Budgets to perform any administrative tasks, or to manage credit as an investor or a manager, or to reconcile transactions as the teller.

To log in to Budgets:

```
amgr login
<local host password>
```

To log out of Budgets:

```
amgr logout
```

6.2 Tutorials

6.2.1 Tutorial on Configuring and Using Budgets in Prepaid Mode

6.2.1.1 Prerequisites

- A working installation of PBS Professional, with at least two accounts that can submit and run jobs at the complex. In our example, the cluster is named Cluster1, and the users are User1 and User2. Substitute in your own names for the cluster and the users when going through the tutorial.
- Install Budgets: follow the instructions in section 2.4, "Prerequisites", on page 31, choose a configuration from section 2.2, "Recommended Configurations", on page 29, and install Budgets according to section 2.6, "Installation Steps for Default Location", on page 37.
- Create test accounts: In addition to an administrator account, you will need investor, manager, and job submitter accounts. Log in as root:

```
adduser Investor1
passwd Investor1 <password>
adduser Manager1
passwd Manager1 <password>
adduser User1
passwd User1 <password>
adduser User2
passwd User2 <password>
```

6.2.1.2 Tutorial Steps to Configure Budgets

6.2.1.2.i Create Periods

1. Log in as, or switch user to pbsadmin:

   ```
   su pbsadmin
   ```

2. Log into `amgr`:

   ```
   amgr login
   ```

3. Create a parent period named "2022" representing the year 2022. See section 1.7.1, "Periods, Allocation Periods, Billing Periods", on page 18.

   ```
   amgr add period -n 2022 -S 2022-01-01 -E 2022-12-31
   ```

4. Create a child period named "2022.Q2" that represents the second quarter of the year 2022, and add it to the parent period:

   ```
   amgr add period -n 2022.Q2 -S 2022-10-01 -E 2022-12-31 -p 2022
   ```

   For the purposes of this tutorial, we are somewhere in the second quarter of 2022, so that this is the default period.

6.2.1.2.ii Add PBS Complex to Budgets

5. Add a cluster named Cluster1 to represent your PBS complex:

   ```
   amgr add cluster -n Cluster1
   ```

6. Deactivate Cluster1:

   ```
   amgr update cluster -n Cluster1 -a False
   ```

7. List Cluster1:

   ```
   amgr ls cluster -a False
   ```

8. Activate Cluster1:

   ```
   amgr update cluster -n Cluster1 -a True
   ```

9. List Cluster1 again:

   ```
   amgr ls cluster
   ```

6.2.1.2.iii Create Standard Service Unit

10. Create a standard service unit named "cpu_hrs" to represent CPU hours. See section 1.7.2, "Service Units", on page 18. The service unit name must be identical to the one that is configured in the formulas section of the `am_hook.json` configuration file:

    ```
    amgr add serviceunit -n cpu_hrs -d "CPU Hours"
    ```

6.2.1.2.iv Add Users to Budgets

11. Add a user who will be a group manager, a user who will be an investor, and users User1 and User2 who will submit jobs. When you add a user, you must assign a role, an accounting policy, and at least one cluster:

    ```
    amgr add user -n Manager1 -r manager -A begin_period -c Cluster1
    amgr add user -n Investor1 -r investor -A begin_period -c Cluster1
    amgr add user -n User1 -r user -A begin_period -c Cluster1
    amgr add user -n User2 -r user -A begin_period -c Cluster1
    ```

6.2.1.2.v Create Group

12. Create a group named "test_group" with *manager* Manager1 and *investor* Investor1:

    ```
    amgr add group -n test_group -M Manager1 -I Investor1
    ```

6.2.1.2.vi Associate Job Submitters with Group

13. Associate User1 and User2 with the group test_group:

    ```
    amgr update user -n User1 -h + test_group
    amgr update user -n User2 -h + test_group
    ```

6.2.1.2.vii Create Project and Give It Cluster and User

14. Create a project named "P1" with accounting policy *begin_period*, cluster Cluster1, user User1 and group test_group:

    ```
    amgr add project -n P1 -A begin_period -c Cluster1 -u User1 -h test_group
    ```

6.2.1.2.viii Invest in Group

15. Log in as Investor1:

    ```
    amgr login
    ```

16. Invest 10000 CPU hours in test_group:

    ```
    amgr deposit group -n test_group -s cpu_hrs 10000
    ```

6.2.1.2.ix Deposit Service Units to Project

17. Log in as Manager1:

    ```
    amgr login
    ```

18. Allocate 1200 CPU hours from test_group to project P1 for the period 2022.Q2:

    ```
    amgr deposit project -n P1 -s cpu_hrs 1200.00 -p 2022.Q2 -h test_group
    ```

19. Check the balance of project P1 for the 2022.Q2 period:

    ```
    amgr checkbalance project -n P1 -p 2022.Q2
    ```

20. Withdraw 200 CPU hours from project P1 for the period 2022.Q2:

    ```
    amgr withdraw project -n P1 -s cpu_hrs 200.00 -p 2022.Q2 -h test_group
    ```

6.2.1.2.x Deposit Service Units to Users

21. Allocate 1300 CPU hours from test_group to user User1 for the period 2022.Q2:

    ```
    amgr deposit user -n User1 -s cpu_hrs 1300.00 -p 2022.Q2 -h test_group
    ```

22. Allocate 1400 CPU hours from test_group to user User2 for the period 2022.Q2:

    ```
    amgr deposit user -n User2 -s cpu_hrs 1400.00 -p 2022.Q2 -h test_group
    ```

6.2.1.3 Tutorial Steps to Use Budgets

6.2.1.3.i Run User Job

23. Log in as User1.
24. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to user User1:
    ```
    qsub -lwalltime=00:02:00 -- /bin/sleep 10
    ```
25. Check the credit balance for user User1. It will have decreased:
    ```
    amgr checkbalance user -n User1 -p 2022.Q2
    ```
26. After the job is finished, check the balance again. You should see that the unused amount has been returned:
    ```
    amgr checkbalance user -n User1 -p 2022.Q2
    ```

6.2.1.3.ii Run Project Job

27. Run a sleep job for 36 seconds with a walltime of 1 hour, and charge it to project P1:
    ```
    qsub -P P1 -lwalltime=01:00:00 -- /bin/sleep 36
    ```
28. To see the job running:
    ```
    qstat -sw
    ```
29. Log in as Manager1.
30. Check the credit balance for project P1. It will have decreased:
    ```
    amgr checkbalance project -n P1 -p 2022.Q2
    ```
31. After the job is finished, check the balance again. You should see that the unused amount has been returned:
    ```
    amgr checkbalance project -n P1 -p 2022.Q2
    ```

6.2.1.3.iii Non-project User Tries to Run Project Job

32. Log in as User2.
33. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to user User2:
    ```
    qsub -lwalltime=00:02:00 --/bin/sleep 10
    ```
34. Check the credit balance for user User2. It will have decreased:
    ```
    amgr checkbalance user -n User2 -p 2022.Q2
    ```
35. After the job is finished, check the balance again. You should see that the unused amount has been returned:
    ```
    amgr checkbalance user -n User2 -p 2022.Q2
    ```
36. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to project P1:
    ```
    qsub -P P1 -lwalltime=00:02:00 -- /bin/sleep 10
    ```
    This job cannot run, because User2 is not part of project P1.

Example 6-1: Report for all standard service units and current lowest period:
```
amgr report project -n p1
```

6.2.1.3.iv Manager Runs Report on Project

37. Log in as Manager1:

    `amgr login`

38. Get report on project P1:

    `amgr report project -n P1`

    Command output:

```
-----------------------------------------------------------------------
name | period    | serviceunit | opening_balance | total_credits
-----------------------------------------------------------------------
P1   | 2022      | cpu_hrs     | 0.0             | 1000.0

------------------------------------------------------------------------
| total_debits | total_debits_reconciled | total_debits_authorized
------------------------------------------------------------------------
| 0.01         | 1.99                    | 0.0

--------------------------
| net_balance | metadata
--------------------------
| 999.99      | {}
```

6.2.2 Tutorial on Configuring Budgets in Postpaid Mode

6.2.2.1 Prerequisites

- A working installation of PBS Professional, with at least two accounts that can submit and run jobs at the complex. In our example, the cluster is named Cluster1, and the users are User1 and User2. Substitute in your own names for the cluster and the users when going through the tutorial.
- Install Budgets: follow the instructions in section 2.4, “Prerequisites”, on page 31, choose a configuration from section 2.2, “Recommended Configurations”, on page 29, and install Budgets according to section 2.6, “Installation Steps for Default Location”, on page 37.
- Create test accounts: In addition to an administrator account, you will need manager and job submitter accounts. Log in as root:

```
adduser Manager1
passwd Manager1 <password>
adduser User1
passwd User1 <password>
adduser User2
passwd User2 <password>
```

6.2.2.2 Tutorial Steps to Configure Budgets

6.2.2.2.i Create Periods

1. Log in as, or switch user to pbsadmin:

   ```
   su pbsadmin
   ```

2. Log into `amgr`:

   ```
   amgr login
   ```

3. Create a parent period named "2022" representing the year 2022. See section 1.7.1, “Periods, Allocation Periods, Billing Periods”, on page 18.

   ```
   amgr add period -n 2022 -S 2022-01-01 -E 2022-12-31
   ```

   For the purposes of this tutorial, we are somewhere in 2022, so that this is the default period.

6.2.2.2.ii Add PBS Complex to Budgets

4. Add a cluster named Cluster1 to represent your PBS complex:

   ```
   amgr add cluster -n Cluster1
   ```

5. Deactivate Cluster1:

   ```
   amgr update cluster -n Cluster1 -a False
   ```

6. List Cluster1:

   ```
   amgr ls cluster -a False
   ```

7. Activate Cluster1:

   ```
   amgr update cluster -n Cluster1 -a True
   ```

8. List Cluster1 again:

   ```
   amgr ls cluster
   ```

6.2.2.2.iii Create Standard Service Unit

9. Create a standard service unit named "cpu_hrs" to represent CPU hours. See section 1.7.2, “Service Units”, on page 18. The service unit name must be identical to the one that is configured in the formulas section of the `am_hook.json` configuration file:

   ```
   amgr add serviceunit -n cpu_hrs -d "CPU Hours"
   ```

6.2.2.2.iv Add Users to Budgets

10. Add a user who will be a group manager, and users User1 and User2 who will submit jobs. When you add a user, you must assign a role, an accounting policy, and at least one cluster:

    ```
    amgr add user -n Manager1 -r manager -A begin_period -c Cluster1
    amgr add user -n User1 -r user -A begin_period -c Cluster1
    amgr add user -n User2 -r user -A begin_period -c Cluster1
    ```

6.2.2.2.v Create Group

11. Create a group named "test_group" with *manager* Manager1:

    ```
    amgr add group -n test_group -M Manager1
    ```

6.2.2.2.vi Associate Job Submitters with Group

12. Associate User1 and User2 with the group test_group:

```
amgr update user -n User1 -h + test_group
amgr update user -n User2 -h + test_group
```

6.2.2.2.vii Create Project and Give It Cluster and User

13. Create a project named "P1" with accounting policy *begin_period*, cluster Cluster1, user User1 and group test_group:

```
amgr add project -n P1 -A begin_period -c Cluster1 -u User1 -h test_group
```

6.2.2.3 Tutorial Steps to Use Budgets

6.2.2.3.i Run User Job

14. Log in as User1.
15. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to user User1:

```
qsub -lwalltime=00:02:00 -- /bin/sleep 10
```

16. After the job is finished, check the credit used by user User1. It will have increased:

```
amgr checkbalance user -n User1 -p 2022
```

6.2.2.3.ii Run Project Job

17. Run a sleep job for 36 seconds with a walltime of 1 hour, and charge it to project P1:

```
qsub -P P1 -lwalltime=01:00:00 -- /bin/sleep 36
```

18. To see the job running:

```
qstat -sw
```

19. Log in as Manager1.
20. After the job is finished, check the credit used by project P1. It will have increased:

```
amgr checkbalance project -n P1 -p 2022
```

6.2.2.3.iii Non-project User Tries to Run Project Job

21. Log in as User2.
22. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to user User2:

```
qsub -lwalltime=00:02:00 --/bin/sleep 10
```

23. After the job is finished, check the credit used by user User2. It will have increased:

```
amgr checkbalance user -n User2 -p 2022
```

24. Run a sleep job for 10 seconds with a walltime of 2 minutes, and charge it to project P1:

```
qsub -P P1 -lwalltime=00:02:00 -- /bin/sleep 10
```

    This job cannot run, because User2 is not part of project P1.

6.2.2.3.iv Manager Runs Report on Project

25. Log in as Manager1:

    ```
    amgr login
    ```

26. Get report on project P1:

    ```
    amgr report project -n P1
    ```

    Command output:

    ```
    ------------------------------------------------------------------------
    name    | period    | serviceunit    | total_outstanding    | metadata
    ------------------------------------------------------------------------
    P1      | 2022      | cpu_hrs        | 000.01               | {}
    ```

Index

Index