About Tractor



Tractor distributes tasks to a farm of execution servers. It manages large queues of concurrent jobs from many users, enforcing dependencies and scheduling policies.

Tractor is installed locally by administrators at each customer studio, and all of the running components and job data are private to each studio. An on-site Tractor farm can be extended to include cloud nodes by using Virtual Private Network techniques to extend the local network to the cloud machines.

Tractor is designed to be a compact system, relative to the complex role that it plays:

- Tractor-Engine: The central job queue manager, a self-contained single install. Low overhead operation with engineering emphasis on high throughput job distribution for large farms.
- Tractor-Blade: Simplified "plug-and-play" deployment of new execution servers on any size farm.
- Tractor-Dashboard Web UI for ubiquitous access to job feedback and farm status.
- Scripting and command-line tools for wranglers and advanced users.
- Component architecture based on common web technologies.

Tractor can drive all of the computational tools used in modern VFX and animation pipelines. It is used to run everything from rendering and compositing to simulation, transcoding, archiving, database updates, code builds, online asset delivery and notifications. Tractor can launch almost any executable available on your Linux, Windows, and macOS systems.

A tractor **Job** describes what needs to get done. Jobs are comprised of **Tasks**, which represent the steps required to complete the job. A job definition describes the resources that are required for each task, and whether the task will run serially or in parallel with other tasks in the job. Jobs can be created by hand, by scripts, by the supplied Python API, or by plug-in job generators in content-creation applications.

Jobs are submitted to the **Engine**, the central process that maintains the job queue. Tractor-engine selects an appropriate Tractor **Blade** execution node on the farm for each command, using an abstract keyword matching scheme and resource limit restrictions. The engine also takes care of enforcing dependencies between tasks, as well as finding opportunities for parallel execution, allowing different jobs or independent parts of a single job to run on different servers concurrently. Tractor provides flexible schemes for job prioritization, compute resource access controls, dynamic server availability, and user limit policies.

The Tractor **Dashboard** web interface provides users and administrators with simple feedback on job status, errors, and overall system status. It also provides drill-down access to detailed information on job structure, attributes, and output logs, as well as detailed data on the blades. The **Tractor Query** to ols provide access to more complex relational queries into the job queue and past execution data. There are query tool interfaces in the web Dashboard, or through a Python API, or on the command line to provide wranglers with quick diagnostics or to build custom reports.

Tractor components communicate using common web-based technologies such as HTTP, JSON, and AJAX. Studios can develop their own custom extensions or external applications that communicate with Tractor using these same standard tools.

Tractor was developed with efficient large studio environments in mind, and as a result it is intentionally light-weight and single purpose. For example, Tractor does not undertake asset management roles, though asset management tasks can be added to Tractor jobs. Typically Tractor jobs are authored assuming that input files for tasks are located on shared file servers at the studio, and that administrators have configured the network to allow commands launched by each Tractor Blade to have access to those shared files.