readme.md for @push.rocks/taskbuffer

> **Modern TypeScript task orchestration and service lifecycle management with constraint-based concurrency, smart buffering, scheduling, health checks, and real-time event streaming**

npm version TypeScript License: MIT

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

□□ Features

- □□ **Type-Safe Task Management** — Full TypeScript support with generics and type inference
- □□ **Constraint-Based Concurrency** — Per-key mutual exclusion, group concurrency limits, cooldown enforcement, sliding-window rate limiting, and result sharing via `TaskConstraintGroup`
- □□ **Real-Time Progress Tracking** — Step-based progress with percentage weights
- ⚡ **Smart Buffering** — Intelligent request debouncing and batching
- □ **Cron Scheduling** — Schedule tasks with cron expressions
- □□ **Task Chains & Parallel Execution** — Sequential and parallel task orchestration
- □□ **Labels** — Attach arbitrary `Record<string, string>` metadata (userId, tenantId, etc.) for multi-tenant filtering
- □□ **Push-Based Events** — rxjs `Subject<ITaskEvent>` on every Task and TaskManager for real-time state change notifications

- □□ **Error Handling** — Configurable error propagation with `catchErrors`, error tracking, and clear error state
- □□ **Service Lifecycle Management** — `Service` and `ServiceManager` for long-running components (databases, servers, queues) with health checks, auto-restart, dependency ordering, and instance access
- □□ **Web Component Dashboard** — Built-in Lit-based dashboard for real-time task visualization
- □□ **Distributed Coordination** — Abstract coordinator for multi-instance task deduplication

□□ Installation

```
pnpm add @push.rocks/taskbuffer
# or
npm install @push.rocks/taskbuffer
```

□□ Quick Start

Basic Task

```
import { Task } from '@push.rocks/taskbuffer';

const greetTask = new Task({
  name: 'Greet',
  taskFunction: async (name) => {
    return `Hello, ${name}!`;
  },
});

const result = await greetTask.trigger('World');
console.log(result); // "Hello, World!"
```

Task with Typed Data □□

Every task can carry a typed data bag — perfect for constraint matching, routing, and metadata:

```
const task = new Task<undefined, [], { domain: string; priority: number }>({
  name: 'update-dns',
  data: { domain: 'example.com', priority: 1 },
  taskFunction: async () => {
    // task.data is fully typed here
    console.log(`Updating DNS for ${task.data.domain}`);
  },
});

task.data.domain;   // string — fully typed
task.data.priority; // number — fully typed
```

Task with Steps & Progress □□

```
const deployTask = new Task({
  name: 'Deploy',
  steps: [
    { name: 'build', description: 'Building app', percentage: 30 },
    { name: 'test', description: 'Running tests', percentage: 20 },
    { name: 'deploy', description: 'Deploying to server', percentage: 40 },
    { name: 'verify', description: 'Verifying deployment', percentage: 10 },
  ] as const,
  taskFunction: async () => {
    deployTask.notifyStep('build');
    await buildApp();

    deployTask.notifyStep('test');
    await runTests();

    deployTask.notifyStep('deploy');
    await deployToServer();

    deployTask.notifyStep('verify');
    await verifyDeployment();

    return 'Deployment successful!';
  },
});
```

```
await deployTask.trigger();
console.log(deployTask.getProgress()); // 100
console.log(deployTask.getStepsMetadata()); // Step details with status
```

> **Note:** `notifyStep()` is fully type-safe — TypeScript only accepts step names you declared in the `steps` array when you use `as const`.

□□ Task Constraints — Concurrency, Mutual Exclusion & Cooldowns

`TaskConstraintGroup` is the unified mechanism for controlling how tasks run relative to each other. It replaces older patterns like task runners, blocking tasks, and execution delays with a single, composable, key-based constraint system.

Per-Key Mutual Exclusion

Ensure only one task runs at a time for a given key (e.g. per domain, per tenant, per resource):

```
import { Task, TaskManager, TaskConstraintGroup } from '@push.rocks/taskbuffer';

const manager = new TaskManager();

// Only one DNS update per domain at a time
const domainMutex = new TaskConstraintGroup<{ domain: string }>({
  name: 'domain-mutex',
  maxConcurrent: 1,
  constraintKeyForExecution: (task, input?) => task.data.domain,
});

manager.addConstraintGroup(domainMutex);

const task1 = new Task<undefined, [], { domain: string }>({
```

```
  name: 'update-a.com',
  data: { domain: 'a.com' },
  taskFunction: async () => { /* update DNS for a.com */ },
});

const task2 = new Task<undefined, [], { domain: string }>({
  name: 'update-a.com-2',
  data: { domain: 'a.com' },
  taskFunction: async () => { /* another update for a.com */ },
});

manager.addTask(task1);
manager.addTask(task2);

// task2 waits until task1 finishes (same domain key)
await Promise.all([
  manager.triggerTask(task1),
  manager.triggerTask(task2),
]);
```

Group Concurrency Limits

Cap how many tasks can run concurrently across a group:

```
// Max 3 DNS updaters running globally at once
const dnsLimit = new TaskConstraintGroup<{ group: string }>({
  name: 'dns-concurrency',
  maxConcurrent: 3,
  constraintKeyForExecution: (task) =>
    task.data.group === 'dns' ? 'dns' : null, // null = skip constraint
});

manager.addConstraintGroup(dnsLimit);
```

Cooldowns (Rate Limiting)

Enforce a minimum time gap between consecutive executions for the same key:

```
// No more than one API call per domain every 11 seconds
const rateLimiter = new TaskConstraintGroup<{ domain: string }>({
  name: 'api-rate-limit',
  maxConcurrent: 1,
  cooldownMs: 11000,
  constraintKeyForExecution: (task) => task.data.domain,
});

manager.addConstraintGroup(rateLimiter);
```

Global Concurrency Cap

Limit total concurrent tasks system-wide:

```
const globalCap = new TaskConstraintGroup({
  name: 'global-cap',
  maxConcurrent: 10,
  constraintKeyForExecution: () => 'all', // same key = shared limit
});

manager.addConstraintGroup(globalCap);
```

Composing Multiple Constraints

Multiple constraint groups stack — a task only runs when **all** applicable constraints allow it:

```
manager.addConstraintGroup(globalCap);     // max 10 globally
manager.addConstraintGroup(domainMutex);   // max 1 per domain
manager.addConstraintGroup(rateLimiter);   // 11s cooldown per domain

// A task must satisfy ALL three constraints before it starts
await manager.triggerTask(dnsTask);
```

Selective Constraints

Return `null` from `constraintKeyForExecution` to exempt a task from a constraint group:

```
const constraint = new TaskConstraintGroup<{ priority: string }>({
  name: 'low-priority-limit',
  maxConcurrent: 2,
  constraintKeyForExecution: (task) =>
    task.data.priority === 'low' ? 'low-priority' : null, // high priority tasks skip this
constraint
});
```

Input-Aware Constraints □□

The `constraintKeyForExecution` function receives both the **task** and the **runtime input** passed to `trigger(input)`. This means the same task triggered with different inputs can be constrained independently:

```
const extractTLD = (domain: string) => {
  const parts = domain.split('.');
  return parts.slice(-2).join('.');
};

// Same TLD → serialized. Different TLDs → parallel.
const tldMutex = new TaskConstraintGroup({
  name: 'tld-mutex',
  maxConcurrent: 1,
  constraintKeyForExecution: (task, input?: string) => {
    if (!input) return null;
    return extractTLD(input); // "example.com", "other.org", etc.
  },
});

manager.addConstraintGroup(tldMutex);

// These two serialize (same TLD "example.com")
const p1 = manager.triggerTaskConstrained(getCert, 'app.example.com');
const p2 = manager.triggerTaskConstrained(getCert, 'api.example.com');

// This runs in parallel (different TLD "other.org")
const p3 = manager.triggerTaskConstrained(getCert, 'my.other.org');
```

You can also combine `task.data` and `input` for composite keys:

```
const providerDomain = new TaskConstraintGroup<{ provider: string }>({
  name: 'provider-domain',
  maxConcurrent: 1,
  constraintKeyForExecution: (task, input?: string) => {
    return `${task.data.provider}:${input || 'default'}`;
  },
});
```

Pre-Execution Check with `shouldExecute`

The `shouldExecute` callback runs right before a queued task executes. If it returns `false`, the task is skipped and its promise resolves with `undefined`. This is perfect for scenarios where a prior execution's outcome makes subsequent queued tasks unnecessary:

```
const certCache = new Map<string, string>();

const certConstraint = new TaskConstraintGroup({
  name: 'cert-mutex',
  maxConcurrent: 1,
  constraintKeyForExecution: (task, input?: string) => {
    if (!input) return null;
    return extractTLD(input);
  },
  shouldExecute: (task, input?: string) => {
    if (!input) return true;
    // Skip if a wildcard cert already covers this TLD
    return certCache.get(extractTLD(input)) !== 'wildcard';
  },
});

const getCert = new Task({
  name: 'get-certificate',
  taskFunction: async (domain: string) => {
    const cert = await acme.getCert(domain);
    if (cert.isWildcard) certCache.set(extractTLD(domain), 'wildcard');
    return cert;
  },
});
```

```
manager.addConstraintGroup(certConstraint);
manager.addTask(getCert);

const r1 = manager.triggerTaskConstrained(getCert, 'app.example.com'); // runs, gets wildcard
const r2 = manager.triggerTaskConstrained(getCert, 'api.example.com'); // queued → skipped!
const r3 = manager.triggerTaskConstrained(getCert, 'my.other.org');    // parallel (different
TLD)

const [cert1, cert2, cert3] = await Promise.all([r1, r2, r3]);
// cert2 === undefined (skipped because wildcard already covers example.com)
```

`shouldExecute` semantics:

- Runs right before execution (after slot acquisition, before `trigger()`)
- Also checked on immediate (non-queued) triggers
- Returns `false` → skip execution, deferred resolves with `undefined`
- Can be async (return `Promise<boolean>`)
- Has closure access to external state modified by prior executions
- If multiple constraint groups have `shouldExecute`, **all** must return `true`

Sliding Window Rate Limiting

Enforce "N completions per time window" with burst capability. Unlike `cooldownMs` (which forces even spacing between executions), `rateLimit` allows bursts up to the cap, then blocks until the window slides:

```
// Let's Encrypt style: 300 new orders per 3 hours
const acmeRateLimit = new TaskConstraintGroup({
  name: 'acme-rate',
  constraintKeyForExecution: () => 'acme-account',
  rateLimit: {
    maxPerWindow: 300,
    windowMs: 3 * 60 * 60 * 1000, // 3 hours
  },
});

manager.addConstraintGroup(acmeRateLimit);

// All 300 can burst immediately. The 301st waits until the oldest
// completion falls out of the 3-hour window.
for (const domain of domains) {
```

```
  manager.triggerTaskConstrained(certTask, { domain });
}
```

Compose multiple rate limits for layered protection:

```
// Per-domain weekly cap AND global order rate
const perDomainWeekly = new TaskConstraintGroup({
  name: 'per-domain-weekly',
  constraintKeyForExecution: (task, input) => input.registeredDomain,
  rateLimit: { maxPerWindow: 50, windowMs: 7 * 24 * 60 * 60 * 1000 },
});

const globalOrderRate = new TaskConstraintGroup({
  name: 'global-order-rate',
  constraintKeyForExecution: () => 'global',
  rateLimit: { maxPerWindow: 300, windowMs: 3 * 60 * 60 * 1000 },
});

manager.addConstraintGroup(perDomainWeekly);
manager.addConstraintGroup(globalOrderRate);
```

Combine with `maxConcurrent` and `cooldownMs` for fine-grained control:

```
const throttled = new TaskConstraintGroup({
  name: 'acme-throttle',
  constraintKeyForExecution: () => 'acme',
  maxConcurrent: 5,          // max 5 concurrent requests
  cooldownMs: 1000,           // 1s gap after each completion
  rateLimit: {
    maxPerWindow: 300,
    windowMs: 3 * 60 * 60 * 1000,
  },
});
```

Result Sharing — Deduplication for Concurrent Requests

When multiple callers request the same resource concurrently, `resultSharingMode: 'share-latest'` ensures only one execution occurs. All queued waiters receive the same result:

```
const certMutex = new TaskConstraintGroup({
  name: 'cert-per-tld',
  constraintKeyForExecution: (task, input) => extractTld(input.domain),
  maxConcurrent: 1,
  resultSharingMode: 'share-latest',
});

manager.addConstraintGroup(certMutex);

const certTask = new Task({
  name: 'obtain-cert',
  taskFunction: async (input) => {
    return await acmeClient.obtainWildcard(input.domain);
  },
});
manager.addTask(certTask);

// Three requests for *.example.com arrive simultaneously
const [cert1, cert2, cert3] = await Promise.all([
  manager.triggerTaskConstrained(certTask, { domain: 'api.example.com' }),
  manager.triggerTaskConstrained(certTask, { domain: 'www.example.com' }),
  manager.triggerTaskConstrained(certTask, { domain: 'mail.example.com' }),
]);

// Only ONE ACME request was made.
// cert1 === cert2 === cert3 — all callers got the same cert object.
```

Result sharing semantics:

- `shouldExecute` is NOT called for shared results (the task's purpose was already fulfilled)
- Error results are NOT shared — queued tasks execute independently after a failure
- `lastResults` persists until `reset()` — for time-bounded sharing, use `shouldExecute` to control staleness
- Composable with rate limiting: rate-limited waiters get shared results without waiting for the window

How It Works

When you trigger a task through `TaskManager` (via `triggerTask`, `triggerTaskByName`, `addExecuteRemoveTask`, or cron), the manager:

1. Evaluates all registered constraint groups against the task and input
2. If no constraints apply (all matchers return `null`) → checks `shouldExecute` → runs or skips
3. If all applicable constraints have capacity → acquires slots → checks `shouldExecute` → runs or skips
4. If any constraint blocks → enqueues the task; when a running task completes, the queue is drained
5. Cooldown/rate-limit-blocked tasks auto-retry after the shortest remaining delay expires
6. Queued tasks check for shared results first (if any group has `resultSharingMode: 'share-latest'`)
7. Queued tasks re-check `shouldExecute` when their turn comes — stale work is automatically pruned

□□ Core Concepts

Task Buffering — Intelligent Request Management

Prevent overwhelming your system with rapid-fire requests:

```
const apiTask = new Task({
  name: 'APIRequest',
  buffered: true,
  bufferMax: 5, // Maximum 5 concurrent executions
  taskFunction: async (endpoint) => {
    return await fetch(endpoint).then((r) => r.json());
  },
});

// Rapid fire 100 calls — only bufferMax execute concurrently
for (let i = 0; i < 100; i++) {
  apiTask.trigger(`/api/data/${i}`);
}
```

Buffer Behavior:

- First `bufferMax` calls execute immediately
- Additional calls are queued
- When buffer is full, new calls overwrite the last queued item
- Perfect for real-time data streams where only recent data matters

Task Chains — Sequential Workflows □□

Build complex workflows with automatic data flow between tasks:

```
import { Taskchain } from '@push.rocks/taskbuffer';

const fetchTask = new Task({
  name: 'Fetch',
  taskFunction: async () => {
    const res = await fetch('/api/data');
    return res.json();
  },
});

const transformTask = new Task({
  name: 'Transform',
  taskFunction: async (data) => {
    return data.map((item) => ({ ...item, transformed: true }));
  },
});

const saveTask = new Task({
  name: 'Save',
  taskFunction: async (transformedData) => {
    await database.save(transformedData);
    return transformedData.length;
  },
});

const pipeline = new Taskchain({
  name: 'DataPipeline',
  taskArray: [fetchTask, transformTask, saveTask],
});

const savedCount = await pipeline.trigger();
console.log(`Saved ${savedCount} items`);
```

Taskchain also supports dynamic mutation:

```
pipeline.addTask(newTask);      // Append to chain
pipeline.removeTask(oldTask);   // Remove by reference (returns boolean)
pipeline.shiftTask();           // Remove & return first task
```

Error context is rich — a chain failure includes the chain name, failing task name, task index, and preserves the original error as `.cause`.

Parallel Execution — Concurrent Processing ⚡

Execute multiple tasks simultaneously:

```
import { Taskparallel } from '@push.rocks/taskbuffer';

const parallel = new Taskparallel({
  taskArray: [emailTask, smsTask, pushNotificationTask, webhookTask],
});

await parallel.trigger(notificationData);
```

Debounced Tasks — Smart Trigger Coalescing □□

Coalesce rapid triggers into a single execution after a quiet period:

```
import { TaskDebounced } from '@push.rocks/taskbuffer';

const searchTask = new TaskDebounced({
  name: 'Search',
  debounceTimeInMillis: 300,
  taskFunction: async (query) => {
    return await searchAPI(query);
  },
});

// Rapid calls — only the last triggers after 300ms of quiet
```

```
searchTask.trigger('h');
searchTask.trigger('he');
searchTask.trigger('hel');
searchTask.trigger('hello'); // ← this one fires
```

TaskOnce — Single-Execution Guard

Ensure a task only runs once, regardless of how many times it's triggered:

```
import { TaskOnce } from '@push.rocks/taskbuffer';

const initTask = new TaskOnce({
  name: 'Init',
  taskFunction: async () => {
    await setupDatabase();
    console.log('Initialized!');
  },
});

await initTask.trigger(); // Runs
await initTask.trigger(); // No-op
await initTask.trigger(); // No-op
console.log(initTask.hasTriggered); // true
```

□□ Labels — Multi-Tenant Task Filtering

Attach arbitrary key-value labels to any task for filtering, grouping, or multi-tenant isolation:

```
const task = new Task({
  name: 'ProcessOrder',
  labels: { userId: 'u-42', tenantId: 'acme-corp', priority: 'high' },
  taskFunction: async (order) => {
    /* ... */
  },
});
```

```
// Manipulate labels at runtime
task.setLabel('region', 'eu-west');
task.getLabel('userId'); // 'u-42'
task.hasLabel('tenantId', 'acme-corp'); // true
task.removeLabel('priority'); // true

// Labels are included in metadata snapshots
const meta = task.getMetadata();
console.log(meta.labels); // { userId: 'u-42', tenantId: 'acme-corp', region: 'eu-west' }
```

Filtering Tasks by Label in TaskManager

```
const manager = new TaskManager();
manager.addTask(orderTask1); // labels: { tenantId: 'acme' }
manager.addTask(orderTask2); // labels: { tenantId: 'globex' }
manager.addTask(orderTask3); // labels: { tenantId: 'acme' }

const acmeTasks = manager.getTasksByLabel('tenantId', 'acme');
// → [orderTask1, orderTask3]

const acmeMetadata = manager.getTasksMetadataByLabel('tenantId', 'acme');
// → [ITaskMetadata, ITaskMetadata]
```

□□ Push-Based Events — Real-Time Task Lifecycle

Every `Task` exposes an rxjs `Subject<ITaskEvent>` that emits events as the task progresses through its lifecycle:

```
import type { ITaskEvent } from '@push.rocks/taskbuffer';

const task = new Task({
  name: 'DataSync',
  steps: [
```

```
    { name: 'fetch', description: 'Fetching data', percentage: 50 },
    { name: 'save', description: 'Saving data', percentage: 50 },
  ] as const,
  taskFunction: async () => {
    task.notifyStep('fetch');
    const data = await fetchData();
    task.notifyStep('save');
    await saveData(data);
  },
});

// Subscribe to individual task events
task.eventSubject.subscribe((event: ITaskEvent) => {
  console.log(`[${event.type}] ${event.task.name} @ ${new
Date(event.timestamp).toISOString()}`);
  if (event.type === 'step') console.log(`  Step: ${event.stepName}`);
  if (event.type === 'failed') console.log(`  Error: ${event.error}`);
});

await task.trigger();
// [started] DataSync @ 2025-01-26T...
// [step] DataSync @ 2025-01-26T...
//   Step: fetch
// [step] DataSync @ 2025-01-26T...
//   Step: save
// [completed] DataSync @ 2025-01-26T...
```

Event Types

Type	When	Extra Fields
`'started'`	Task begins execution	—
`'step'`	`notifyStep()` is called	`stepName`
`'completed'`	Task finishes successfully	—
`'failed'`	Task throws an error	`error` (message string)

Every event includes a full `ITaskMetadata` snapshot (including labels) at the time of emission.

Aggregated Events on TaskManager

`TaskManager` automatically aggregates events from all added tasks into a single `taskSubject`:

```
const manager = new TaskManager();
manager.addTask(syncTask);
manager.addTask(reportTask);
manager.addTask(cleanupTask);

// Single subscription for ALL task events
manager.taskSubject.subscribe((event) => {
  sendToMonitoringDashboard(event);
});

// Events stop flowing for a task after removal
manager.removeTask(syncTask);
```

`manager.stop()` automatically cleans up all event subscriptions.

□□ Error Handling

By default, `trigger()` **rejects** when the task function throws — errors propagate naturally:

```
const task = new Task({
  name: 'RiskyOp',
  taskFunction: async () => {
    throw new Error('something broke');
  },
});

try {
  await task.trigger();
} catch (err) {
  console.error(err.message); // "something broke"
}
```

Swallowing Errors with `catchErrors`

Set `catchErrors: true` to swallow errors and return `undefined` instead of rejecting:

```
const task = new Task({
  name: 'BestEffort',
  catchErrors: true,
  taskFunction: async () => {
    throw new Error('non-critical');
  },
});

const result = await task.trigger(); // undefined (no throw)
```

Error State Tracking

Regardless of `catchErrors`, the task tracks errors:

```
console.log(task.lastError); // Error object (or undefined)
console.log(task.errorCount); // Number of failures across all runs
console.log(task.getMetadata().status); // 'failed'

task.clearError(); // Resets lastError to undefined (errorCount stays)
```

On a subsequent successful run, `lastError` is automatically cleared.

□□ TaskManager — Centralized Orchestration

```
const manager = new TaskManager();

// Add tasks
manager.addTask(dataProcessor);
manager.addTask(deployTask);

// Schedule with cron expressions
manager.addAndScheduleTask(backupTask, '0 2 * * *'); // Daily at 2 AM
manager.addAndScheduleTask(healthCheck, '*/5 * * * *'); // Every 5 minutes
```

```
// Register constraint groups
manager.addConstraintGroup(globalCap);
manager.addConstraintGroup(perDomainMutex);

// Query metadata
const meta = manager.getTaskMetadata('Deploy');
console.log(meta);
// {
//   name: 'Deploy',
//   status: 'completed',
//   steps: [...],
//   currentProgress: 100,
//   runCount: 3,
//   labels: { env: 'production' },
//   lastError: undefined,
//   errorCount: 0,
//   ...
// }

// All tasks at once
const allMeta = manager.getAllTasksMetadata();

// Scheduled task info
const scheduled = manager.getScheduledTasks();
const nextRuns = manager.getNextScheduledRuns(5);

// Trigger by name (routes through constraints)
await manager.triggerTaskByName('Deploy');

// One-shot: add, execute, collect report, remove
const report = await manager.addExecuteRemoveTask(temporaryTask);
console.log(report);
// {
//   taskName: 'TempTask',
//   startTime: 1706284800000,
//   endTime: 1706284801523,
//   duration: 1523,
//   steps: [...],
```

```
//   stepsCompleted: ['step1', 'step2'],
//   progress: 100,
//   result: any
// }

// Lifecycle
await manager.start(); // Starts cron scheduling + distributed coordinator
await manager.stop(); // Stops scheduling, cleans up event subscriptions
```

Remove Tasks

```
manager.removeTask(task); // Removes from map and unsubscribes event forwarding
manager.descheduleTaskByName('Deploy'); // Remove cron schedule only
```

Remove Constraint Groups

```
manager.removeConstraintGroup('domain-mutex'); // By name
```

□□ Service Lifecycle Management

For long-running components like database connections, HTTP servers, and message queues, taskbuffer provides `Service` and `ServiceManager` — a complete lifecycle management system with health checks, dependency ordering, retry, auto-restart, and typed instance access.

Basic Service — Builder Pattern

```
import { Service, ServiceManager } from '@push.rocks/taskbuffer';

const dbService = new Service<DatabasePool>('Database')
  .critical()
  .withStart(async () => {
    const pool = new DatabasePool({ host: 'localhost', port: 5432 });
    await pool.connect();
    return pool; // stored as service.instance
  })
```

```
  .withStop(async (pool) => {
    await pool.disconnect(); // receives the instance from start
  })
  .withHealthCheck(async (pool) => {
    return await pool.ping(); // receives the instance too
  });

await dbService.start();
dbService.instance!.query('SELECT 1'); // typed access to the pool
await dbService.stop();
```

The `start()` return value is stored as `service.instance` and automatically passed to `stop()` and `healthCheck()` functions — no need for external closures or shared variables.

Service with Dependencies & Health Checks

```
const cacheService = new Service('Redis')
  .optional()
  .withStart(async () => new RedisClient())
  .withStop(async (client) => client.quit())
  .withHealthCheck(async (client) => client.isReady, {
    intervalMs: 10000,           // check every 10s
    timeoutMs: 3000,             // 3s timeout per check
    failuresBeforeDegraded: 3,   // 3 consecutive failures → 'degraded'
    failuresBeforeFailed: 5,     // 5 consecutive failures → 'failed'
    autoRestart: true,           // auto-restart when failed
    maxAutoRestarts: 5,          // give up after 5 restart attempts
    autoRestartDelayMs: 2000,    // start with 2s delay
    autoRestartBackoffFactor: 2, // double delay each attempt
  });

const apiService = new Service('API')
  .critical()
  .dependsOn('Database', 'Redis')
  .withStart(async () => {
    const server = createServer();
    await server.listen(3000);
```

```
    return server;
  })
  .withStop(async (server) => server.close())
  .withStartupTimeout(10000); // fail if start takes > 10s
```

ServiceManager — Orchestration

`ServiceManager` handles dependency-ordered startup, failure isolation, and aggregated health reporting:

```
const manager = new ServiceManager({
  name: 'MyApp',
  startupTimeoutMs: 60000,   // global startup timeout
  shutdownTimeoutMs: 15000,  // per-service shutdown timeout
  defaultRetry: { maxRetries: 3, baseDelayMs: 1000, backoffFactor: 2 },
});

manager.addService(dbService);
manager.addService(cacheService);
manager.addService(apiService);

await manager.start();
// □ Starts Database first, then Redis (parallel with DB since independent),
//    then API (after both deps are running)
// □ If Database (critical) fails → rollback, stop everything, throw
// ⚠□ If Redis (optional) fails → log warning, continue, health = 'degraded'

// Health aggregation
const health = manager.getHealth();
// { overall: 'healthy', services: [...], startedAt: 1706284800000, uptime: 42000 }

// Cascade restart — stops dependents first, restarts target, then restarts dependents
await manager.restartService('Database');

// Graceful reverse-order shutdown
await manager.stop();
```

Subclass Pattern

For complex services, extend `Service` and override the lifecycle hooks:

```
class PostgresService extends Service<Pool> {
  constructor(private config: PoolConfig) {
    super('Postgres');
    this.critical();
  }

  protected async serviceStart(): Promise<Pool> {
    const pool = new Pool(this.config);
    await pool.connect();
    return pool;
  }

  protected async serviceStop(): Promise<void> {
    await this.instance?.end();
  }

  protected async serviceHealthCheck(): Promise<boolean> {
    const result = await this.instance?.query('SELECT 1');
    return result?.rows.length === 1;
  }
}
```

Waiting for Service Readiness

Programmatically wait for a service to reach a specific state:

```
// Wait for the service to be running (with timeout)
await dbService.waitForRunning(10000);

// Wait for any state
await service.waitForState(['running', 'degraded'], 5000);

// Wait for shutdown
await service.waitForStopped();
```

Service Labels

Tag services with metadata for filtering and grouping:

```
const service = new Service('Redis')
  .withLabels({ type: 'cache', env: 'production', region: 'eu-west' })
  .withStart(async () => new RedisClient())
  .withStop(async (client) => client.quit());

// Query by label in ServiceManager
const caches = manager.getServicesByLabel('type', 'cache');
const prodStatuses = manager.getServicesStatusByLabel('env', 'production');
```

Service Events

Every `Service` emits events via an rxjs `Subject<IServiceEvent>`:

```
service.eventSubject.subscribe((event) => {
  console.log(`[${event.type}] ${event.serviceName} → ${event.state}`);
});
// [started] Database → running
// [healthCheck] Database → running
// [degraded] Database → degraded
// [autoRestarting] Database → failed
// [started] Database → running
// [recovered] Database → running
// [stopped] Database → stopped
```

Event Type	When
`'started'`	Service started successfully
`'stopped'`	Service stopped
`'failed'`	Service start failed or health check threshold exceeded
`'degraded'`	Health check failures exceeded `failuresBeforeDegraded`
`'recovered'`	Health check succeeded while in degraded state
`'retrying'`	ServiceManager retrying a failed start attempt
`'healthCheck'`	Health check completed (success or failure)
`'autoRestarting'`	Auto-restart scheduled after health check failure

`ServiceManager.serviceSubject` aggregates events from all registered services.

Service State Machine

```
stopped → starting → running → degraded → failed
  ↑          ↓                    ↓         ↓
  └── stopping ←──────────────────┘─────────┘
                                  (auto-restart)
```

□□ Web Component Dashboard

Visualize your tasks in real-time with the included Lit-based web component:

```
<script type="module">
  import { TaskManager } from '@push.rocks/taskbuffer';
  import '@push.rocks/taskbuffer/dist_ts_web/taskbuffer-dashboard.js';

  const manager = new TaskManager();
  // ... add and schedule tasks ...

  const dashboard = document.querySelector('taskbuffer-dashboard');
  dashboard.taskManager = manager;
  dashboard.refreshInterval = 500; // Poll every 500ms
</script>

<taskbuffer-dashboard></taskbuffer-dashboard>
```

The dashboard provides:

- □□ Real-time progress bars with step indicators
- □□ Task execution history and metadata
- □ Scheduled task information with next-run times
- □□ Light/dark theme support

□□ Distributed Coordination

For multi-instance deployments, extend `AbstractDistributedCoordinator` to prevent duplicate task execution:

```
import { TaskManager, distributedCoordination } from '@push.rocks/taskbuffer';

class RedisCoordinator extends distributedCoordination.AbstractDistributedCoordinator {
  async fireDistributedTaskRequest(request) {
    // Implement leader election / distributed lock via Redis
    return { shouldTrigger: true, considered: true, rank: 1, reason: 'elected', ...request };
  }
  async updateDistributedTaskRequest(request) {
    /* update status */
  }
  async start() {
    /* connect */
  }
  async stop() {
    /* disconnect */
  }
}

const manager = new TaskManager({
  distributedCoordinator: new RedisCoordinator(),
});
```

When a distributed coordinator is configured, scheduled tasks consult it before executing — only the elected instance runs the task.

□□ Advanced Patterns

Pre-Task & After-Task Hooks

Run setup/teardown tasks automatically:

```
const mainTask = new Task({
  name: 'MainWork',
  preTask: new Task({
    name: 'Setup',
    taskFunction: async () => {
      console.log('Setting up...');
```

```
    },
  }),
  afterTask: new Task({
    name: 'Cleanup',
    taskFunction: async () => {
      console.log('Cleaning up...');
    },
  }),
  taskFunction: async () => {
    console.log('Doing work...');
    return 'done';
  },
});

await mainTask.trigger();
// Setting up... → Doing work... → Cleaning up...
```

One-Time Setup Functions

Run an expensive initialization exactly once, before the first execution:

```
const task = new Task({
  name: 'DBQuery',
  taskSetup: async () => {
    const pool = await createConnectionPool();
    return pool; // This becomes `setupValue`
  },
  taskFunction: async (input, pool) => {
    return await pool.query(input);
  },
});

await task.trigger('SELECT * FROM users'); // Setup runs here
await task.trigger('SELECT * FROM orders'); // Setup skipped, pool reused
```

Database Migration Pipeline

```
const migration = new Taskchain({
  name: 'DatabaseMigration',
  taskArray: [backupTask, validateSchemaTask, runMigrationsTask, verifyIntegrityTask],
});

try {
  await migration.trigger();
  console.log('Migration successful!');
} catch (error) {
  // error includes chain name, failing task name, index, and original cause
  console.error(error.message);
  await rollbackTask.trigger();
}
```

Multi-Tenant SaaS Monitoring

Combine labels + events + constraints for a real-time multi-tenant system:

```
const manager = new TaskManager();

// Per-tenant concurrency limit
const tenantLimit = new TaskConstraintGroup<{ tenantId: string }>({
  name: 'tenant-concurrency',
  maxConcurrent: 2,
  constraintKeyForExecution: (task, input?) => task.data.tenantId,
});
manager.addConstraintGroup(tenantLimit);

// Create tenant-scoped tasks
function createTenantTask(tenantId: string, taskName: string, fn: () => Promise<any>) {
  const task = new Task<undefined, [], { tenantId: string }>({
    name: `${tenantId}:${taskName}`,
    data: { tenantId },
    labels: { tenantId },
    taskFunction: fn,
  });
  manager.addTask(task);
  return task;
```

```
}

createTenantTask('acme', 'sync', async () => syncData('acme'));
createTenantTask('globex', 'sync', async () => syncData('globex'));

// Stream events to tenant-specific WebSocket channels
manager.taskSubject.subscribe((event) => {
  const tenantId = event.task.labels?.tenantId;
  if (tenantId) {
    wss.broadcast(tenantId, JSON.stringify(event));
  }
});

// Query tasks for a specific tenant
const acmeTasks = manager.getTasksMetadataByLabel('tenantId', 'acme');
```

□□ API Reference

Classes

Class	Description
`Task<T, TSteps, TData>`	Core task unit with typed data, optional step tracking, labels, and event streaming
`TaskManager`	Centralized orchestrator with constraint groups, scheduling, label queries, and aggregated events
`TaskConstraintGroup<TData>`	Concurrency, mutual exclusion, and cooldown constraints with key-based grouping
`Taskchain`	Sequential task executor with data flow between tasks
`Taskparallel`	Concurrent task executor via `Promise.all()`
`TaskOnce`	Single-execution guard
`TaskDebounced`	Debounced task using rxjs
`TaskStep`	Step tracking unit (internal, exposed via metadata)
`Service<T>`	Long-running component with start/stop lifecycle, health checks, auto-restart, and typed instance access
`ServiceManager`	Service orchestrator with dependency ordering, failure isolation, retry, and health aggregation

Task Constructor Options

Option	Type	Default	Description
`taskFunction`	`ITaskFunction<T>`	*required*	The async function to execute
`name`	`string`	—	Task identifier (required for TaskManager)
`data`	`TData`	`{}`	Typed data bag for constraint matching and routing
`steps`	`ReadonlyArray<{name, description, percentage}>`	—	Step definitions for progress tracking
`buffered`	`boolean`	—	Enable request buffering
`bufferMax`	`number`	—	Max buffered calls
`preTask`	`Task | () => Task`	—	Task to run before
`afterTask`	`Task | () => Task`	—	Task to run after
`taskSetup`	`() => Promise<T>`	—	One-time setup function
`catchErrors`	`boolean`	`false`	Swallow errors instead of rejecting
`labels`	`Record<string, string>`	`{}`	Initial labels

Task Methods

Method	Returns	Description
`trigger(input?)`	`Promise<any>`	Execute the task
`notifyStep(name)`	`void`	Advance to named step (type-safe)
`getProgress()`	`number`	Current progress 0–100
`getStepsMetadata()`	`ITaskStep[]`	Step details with status
`getMetadata()`	`ITaskMetadata`	Full task metadata snapshot
`setLabel(key, value)`	`void`	Set a label
`getLabel(key)`	`string | undefined`	Get a label value
`removeLabel(key)`	`boolean`	Remove a label
`hasLabel(key, value?)`	`boolean`	Check label existence / value
`clearError()`	`void`	Reset `lastError` to undefined

Task Properties

Property	Type	Description
`name`	`string`	Task identifier
`data`	`TData`	Typed data bag
`running`	`boolean`	Whether the task is currently executing
`idle`	`boolean`	Inverse of `running`
`labels`	`Record<string, string>`	Attached labels
`eventSubject`	`Subject<ITaskEvent>`	rxjs Subject emitting lifecycle events
`lastError`	`Error | undefined`	Last error encountered
`errorCount`	`number`	Total error count across all runs
`runCount`	`number`	Total execution count
`lastRun`	`Date | undefined`	Timestamp of last execution

TaskConstraintGroup Constructor Options

Option	Type	Default	Description
`name`	`string`	*required*	Constraint group identifier
`constraintKeyForExecution`	`(task, input?) => string | null`	*required*	Returns key for grouping, or `null` to skip. Receives both the task and runtime input.
`maxConcurrent`	`number`	`Infinity`	Max concurrent tasks per key
`cooldownMs`	`number`	`0`	Minimum ms between completions per key
`shouldExecute`	`(task, input?) => boolean | Promise<boolean>`	—	Pre-execution check. Return `false` to skip; deferred resolves `undefined`.
`rateLimit`	`IRateLimitConfig`	—	Sliding window: `{ maxPerWindow, windowMs }`. Counts running + completed tasks.
`resultSharingMode`	`TResultSharingMode`	`'none'`	`'none'` or `'share-latest'`. Queued tasks get first task's result without executing.

TaskConstraintGroup Methods

Method	Returns	Description
`getConstraintKey(task, input?)`	`string | null`	Get the constraint key for a task + input
`checkShouldExecute(task, input?)`	`Promise<boolean>`	Run the `shouldExecute` callback (defaults to `true`)
`canRun(key)`	`boolean`	Check if a slot is available (considers concurrency, cooldown, and rate limit)
`acquireSlot(key)`	`void`	Claim a running slot
`releaseSlot(key)`	`void`	Release a slot and record completion time + rate-limit timestamp
`getCooldownRemaining(key)`	`number`	Milliseconds until cooldown expires
`getRateLimitDelay(key)`	`number`	Milliseconds until a rate-limit slot opens
`getNextAvailableDelay(key)`	`number`	Max of cooldown + rate-limit delay — unified "when can I run"
`getRunningCount(key)`	`number`	Current running count for key
`recordResult(key, result)`	`void`	Store result for sharing (no-op if mode is `'none'`)
`getLastResult(key)`	`{result, timestamp} | undefined`	Get last shared result for key
`hasResultSharing()`	`boolean`	Whether result sharing is enabled
`reset()`	`void`	Clear all state (running counts, cooldowns, rate-limit timestamps, shared results)

TaskManager Methods

Method	Returns	Description
`addTask(task)`	`void`	Register a task (wires event forwarding)
`removeTask(task)`	`void`	Remove task and unsubscribe events
`getTaskByName(name)`	`Task | undefined`	Look up by name
`triggerTaskByName(name)`	`Promise<any>`	Trigger by name (routes through constraints)
`triggerTask(task)`	`Promise<any>`	Trigger directly (routes through constraints)

Method	Returns	Description
`triggerTaskConstrained(task, input?)`	`Promise<any>`	Core constraint evaluation entry point
`addConstraintGroup(group)`	`void`	Register a constraint group
`removeConstraintGroup(name)`	`void`	Remove a constraint group by name
`addAndScheduleTask(task, cron)`	`void`	Register + schedule
`scheduleTaskByName(name, cron)`	`void`	Schedule existing task
`descheduleTaskByName(name)`	`void`	Remove schedule
`getTaskMetadata(name)`	`ITaskMetadata | null`	Single task metadata
`getAllTasksMetadata()`	`ITaskMetadata[]`	All tasks metadata
`getScheduledTasks()`	`IScheduledTaskInfo[]`	Scheduled task info
`getNextScheduledRuns(limit?)`	`Array<{...}>`	Upcoming scheduled runs
`getTasksByLabel(key, value)`	`Task[]`	Filter tasks by label
`getTasksMetadataByLabel(key, value)`	`ITaskMetadata[]`	Filter metadata by label
`addExecuteRemoveTask(task, opts?)`	`Promise<ITaskExecutionReport>`	One-shot execution with report
`start()`	`Promise<void>`	Start cron + coordinator
`stop()`	`Promise<void>`	Stop cron + clean up subscriptions

TaskManager Properties

Property	Type	Description
`taskSubject`	`Subject<ITaskEvent>`	Aggregated events from all added tasks
`taskMap`	`ObjectMap<Task>`	Internal task registry
`constraintGroups`	`TaskConstraintGroup[]`	Registered constraint groups

Service Builder Methods

Method	Returns	Description
`critical()`	`this`	Mark as critical (startup failure aborts ServiceManager)
`optional()`	`this`	Mark as optional (startup failure is tolerated)
`dependsOn(...names)`	`this`	Declare dependencies by service name

Method	Returns	Description
`withStart(fn)`	`this`	Set start function: `() => Promise<T>`
`withStop(fn)`	`this`	Set stop function: `(instance: T) => Promise<void>`
`withHealthCheck(fn, config?)`	`this`	Set health check: `(instance: T) => Promise<boolean>`
`withRetry(config)`	`this`	Set retry config: `{ maxRetries, baseDelayMs, maxDelayMs, backoffFactor }`
`withStartupTimeout(ms)`	`this`	Per-service startup timeout
`withLabels(labels)`	`this`	Attach key-value labels

Service Methods

Method	Returns	Description
`start()`	`Promise<T>`	Start the service (no-op if already running)
`stop()`	`Promise<void>`	Stop the service (no-op if already stopped)
`checkHealth()`	`Promise<boolean | undefined>`	Run health check manually
`waitForState(target, timeoutMs?)`	`Promise<void>`	Wait for service to reach a state
`waitForRunning(timeoutMs?)`	`Promise<void>`	Wait for `'running'` state
`waitForStopped(timeoutMs?)`	`Promise<void>`	Wait for `'stopped'` state
`getStatus()`	`IServiceStatus`	Full status snapshot
`setLabel(key, value)`	`void`	Set a label
`getLabel(key)`	`string | undefined`	Get a label value
`removeLabel(key)`	`boolean`	Remove a label
`hasLabel(key, value?)`	`boolean`	Check label existence / value

Service Properties

Property	Type	Description
`name`	`string`	Service identifier
`state`	`TServiceState`	Current state (`stopped`, `starting`, `running`, `degraded`, `failed`, `stopping`)

Property	Type	Description
`instance`	`T | undefined`	The value returned from `start()`
`criticality`	`TServiceCriticality`	`'critical'` or `'optional'`
`dependencies`	`string[]`	Dependency names
`labels`	`Record<string, string>`	Attached labels
`eventSubject`	`Subject<IServiceEvent>`	rxjs Subject emitting lifecycle events
`errorCount`	`number`	Total error count
`retryCount`	`number`	Retry attempts during last startup

ServiceManager Methods

Method	Returns	Description
`addService(service)`	`void`	Register a service
`addServiceFromOptions(options)`	`Service<T>`	Create and register from options
`removeService(name)`	`void`	Remove service (throws if others depend on it)
`start()`	`Promise<void>`	Start all services in dependency order
`stop()`	`Promise<void>`	Stop all services in reverse order
`restartService(name)`	`Promise<void>`	Cascade restart with dependents
`getService(name)`	`Service | undefined`	Look up by name
`getServiceStatus(name)`	`IServiceStatus | undefined`	Single service status
`getAllStatuses()`	`IServiceStatus[]`	All service statuses
`getHealth()`	`IServiceManagerHealth`	Aggregated health report
`getServicesByLabel(key, value)`	`Service[]`	Filter services by label
`getServicesStatusByLabel(key, value)`	`IServiceStatus[]`	Filter statuses by label

Exported Types

```
import type {
  // Task types
  ITaskMetadata,
  ITaskExecutionReport,
  ITaskExecution,
```

```
  IScheduledTaskInfo,
  ITaskEvent,
  TTaskEventType,
  ITaskStep,
  ITaskFunction,
  ITaskConstraintGroupOptions,
  IRateLimitConfig,
  TResultSharingMode,
  StepNames,
  // Service types
  IServiceOptions,
  IServiceStatus,
  IServiceEvent,
  IServiceManagerOptions,
  IServiceManagerHealth,
  IRetryConfig,
  IHealthCheckConfig,
  TServiceState,
  TServiceCriticality,
  TServiceEventType,
  TOverallHealth,
} from '@push.rocks/taskbuffer';
```

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #3
Created 2026-03-28 11:13:29 UTC by foss.global Team
Updated 2026-03-28 12:20:15 UTC by foss.global Team