readme.md for @push.rocks/smartvm

A TypeScript module that wraps Amazon's Firecracker VMM to create, configure, and manage lightweight microVMs with a clean, type-safe API.

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.

Install

```
pnpm install @push.rocks/smartvm
```

> ⚡ **Prerequisites**: Firecracker requires a Linux host with KVM support (`/dev/kvm`). Networking features (TAP devices, bridges, NAT) require root privileges.

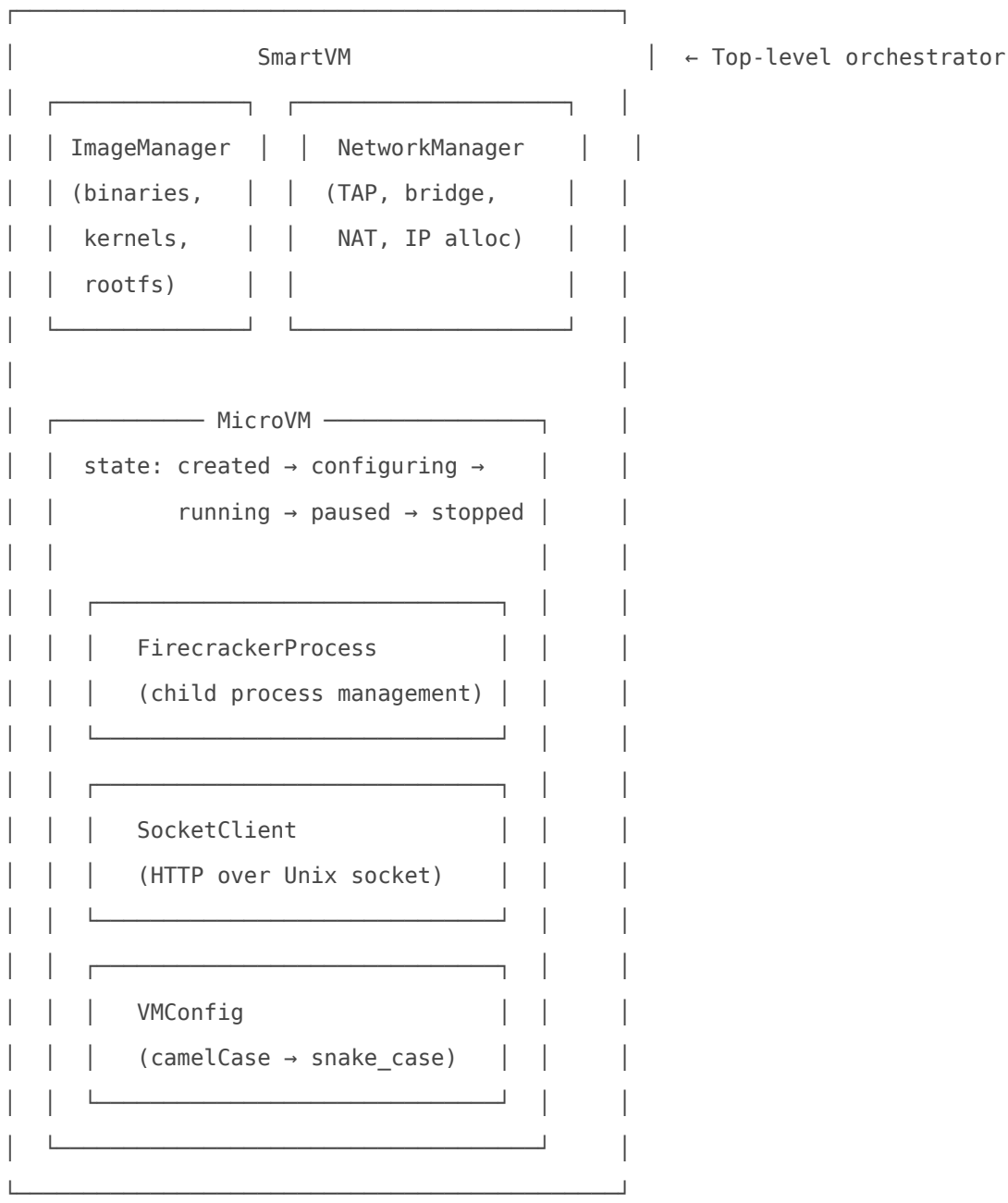
Quick Start

```
import { SmartVM } from '@push.rocks/smartvm';

// 1. Create the orchestrator
const smartvm = new SmartVM({
  dataDir: '/opt/smartvm',         // where binaries, kernels, rootfs are cached
```

```
  firecrackerVersion: 'v1.7.0',  // or omit for latest
  arch: 'x86_64',
});

// 2. Download Firecracker if not already present
await smartvm.ensureBinary();

// 3. Create a MicroVM
const vm = await smartvm.createVM({
  bootSource: {
    kernelImagePath: '/opt/smartvm/kernels/vmlinux',
    bootArgs: 'console=ttyS0 reboot=k panic=1 pci=off',
  },
  machineConfig: {
    vcpuCount: 2,
    memSizeMib: 256,
  },
  drives: [
    {
      driveId: 'rootfs',
      pathOnHost: '/opt/smartvm/rootfs/ubuntu.ext4',
      isRootDevice: true,
      isReadOnly: false,
    },
  ],
  networkInterfaces: [
    { ifaceId: 'eth0' }, // TAP device and MAC auto-generated
  ],
});

// 4. Start it □□
await vm.start();

// 5. Inspect
console.log(vm.state);           // 'running'
console.log(await vm.getInfo()); // Firecracker instance info

// 6. Pause / Resume
await vm.pause();   // state → 'paused'
```

```
await vm.resume();  // state → 'running'

// 7. Stop and clean up
await vm.stop();
await vm.cleanup();
await smartvm.cleanup();
```

Architecture Overview



Firecracker exposes a REST API over a Unix domain socket. This module handles all the plumbing: spawning the process, waiting for the socket, translating your TypeScript config into Firecracker's snake_case API payloads, managing TAP devices, and tearing everything down on exit.

API Reference

`SmartVM` — The Orchestrator

The entry point for everything. Manages binary downloads, VM creation, and global cleanup.

```
import { SmartVM } from '@push.rocks/smartvm';
import type { ISmartVMOptions } from '@push.rocks/smartvm';

const smartvm = new SmartVM({
  dataDir: '/tmp/.smartvm',        // default: /tmp/.smartvm
  firecrackerVersion: 'v1.7.0',  // default: latest from GitHub
  arch: 'x86_64',                // default: x86_64 (also: aarch64)
  firecrackerBinaryPath: '/usr/bin/firecracker', // optional: skip download
  bridgeName: 'svbr0',           // default: svbr0
  subnet: '172.30.0.0/24',       // default: 172.30.0.0/24
});
```

Method	Description
`ensureBinary()`	Downloads Firecracker from GitHub if not cached. Returns path to binary.
`createVM(config)`	Creates a `MicroVM` instance (not yet started). Returns the VM.
`getVM(id)`	Look up an active VM by ID.
`listVMs()`	Returns an array of active VM IDs.
`vmCount`	Number of active VMs.
`stopAll()`	Stops all running/paused VMs in parallel.
`cleanup()`	Stops all VMs, removes TAP devices and bridge.

`MicroVM` — VM Lifecycle

Each VM follows a strict state machine: **created → configuring → running → paused → stopped**.

```
const vm = await smartvm.createVM({
  id: 'my-vm',  // optional, auto-generated UUID if omitted
  bootSource: {
    kernelImagePath: '/path/to/vmlinux',
    bootArgs: 'console=ttyS0 reboot=k panic=1',
    initrdPath: '/path/to/initrd',  // optional
  },
  machineConfig: {
    vcpuCount: 4,
    memSizeMib: 512,
    smt: false,
    cpuTemplate: 'T2',  // optional: C3, T2, T2S, T2CL, T2A, V1N1
    trackDirtyPages: true,
  },
  drives: [
    {
      driveId: 'rootfs',
      pathOnHost: '/path/to/rootfs.ext4',
      isRootDevice: true,
      isReadOnly: false,
      cacheType: 'Unsafe',  // or 'Writeback'
      rateLimiter: {
        bandwidth: { size: 100_000_000, refillTime: 1_000_000_000 },
        ops: { size: 1000, refillTime: 1_000_000_000 },
      },
    },
  ],
  networkInterfaces: [
    {
      ifaceId: 'eth0',
      // hostDevName and guestMac auto-generated if omitted
    },
  ],
  vsock: {
    guestCid: 3,
    udsPath: '/tmp/vsock.sock',
  },
```

```
  balloon: {
    amountMib: 128,
    deflateOnOom: true,
    statsPollingIntervalS: 5,
  },
  mmds: {
    version: 'V2',
    networkInterfaces: ['eth0'],
  },
  logger: {
    logPath: '/tmp/firecracker.log',
    level: 'Debug',
    showLogOrigin: true,
  },
  metrics: {
    metricsPath: '/tmp/firecracker-metrics.fifo',
  },
});
```

Method	Valid States	Description
`start()`	`created`	Spawns Firecracker, applies config, boots the VM
`pause()`	`running`	Pauses VM execution
`resume()`	`paused`	Resumes a paused VM
`stop()`	`running`, `paused`	Graceful shutdown (Ctrl+Alt+Del), then force kill
`cleanup()`	any	Full cleanup: kill process, remove socket, remove TAPs
`getInfo()`	any (after start)	Returns Firecracker instance info
`getVersion()`	any (after start)	Returns Firecracker version
`createSnapshot(params)`	`paused`	Create a VM snapshot
`loadSnapshot(params)`	`created`, `configuring`	Load a VM from snapshot
`setMetadata(data)`	`running`, `paused`	Set MMDS metadata
`getMetadata()`	`running`, `paused`	Get MMDS metadata
`updateDrive(id, path)`	`running`, `paused`	Hot-update a drive path
`updateBalloon(mib)`	`running`, `paused`	Resize the balloon device

Method	Valid States	Description
`getTapDevices()`	any	Returns TAP devices associated with this VM

`ImageManager` — Binary & Image Management

Handles downloading and caching Firecracker binaries, kernels, and rootfs images.

```
const imageManager = smartvm.imageManager;

// Auto-download the latest Firecracker release
const version = await imageManager.getLatestVersion(); // e.g. 'v1.7.0'
const binaryPath = await imageManager.downloadFirecracker(version);

// Download kernel and rootfs images
const kernelPath = await imageManager.downloadKernel(
  'https://example.com/vmlinux-5.10',
  'vmlinux-5.10',
);
const rootfsPath = await imageManager.downloadRootfs(
  'https://example.com/ubuntu-22.04.ext4',
  'ubuntu-22.04.ext4',
);

// Create a blank rootfs or clone an existing one
const blankPath = await imageManager.createBlankRootfs('scratch.ext4', 1024);
const clonePath = await imageManager.cloneRootfs(rootfsPath, 'ubuntu-clone.ext4');
```

Data directory layout:

```
/tmp/.smartvm/
  bin/<version>/firecracker
  bin/<version>/jailer
  kernels/<name>
  rootfs/<name>
  sockets/<vmId>.sock
```

`NetworkManager` — Host Networking

Automatically manages TAP devices, a Linux bridge, and iptables NAT masquerade rules so VMs get internet access out of the box.

```
const networkManager = smartvm.networkManager;

// Manually create a TAP device (usually handled by MicroVM.start())
const tap = await networkManager.createTapDevice('vm-id', 'eth0');
console.log(tap);
// {
//   tapName: 'svvmideth0',
//   guestIp: '172.30.0.2',
//   gatewayIp: '172.30.0.1',
//   subnetMask: '255.255.255.0',
//   mac: '02:a3:b1:c4:d2:e5'
// }

// Generate kernel boot args for the guest
const bootArgs = networkManager.getGuestNetworkBootArgs(tap);
// 'ip=172.30.0.2::172.30.0.1:255.255.255.0::eth0:off'
```

Networking architecture:

- Creates a Linux bridge (default: `svbr0`) with gateway at `.1`
- Each VM gets a TAP device attached to the bridge
- Sequential IP allocation from `.2` onwards
- iptables NAT masquerade for outbound internet
- Deterministic MAC generation (`02:xx:xx:xx:xx:xx` locally-administered)
- TAP names fit Linux's 15-char IFNAMSIZ limit

`VMConfig` — Config Transformer

Converts your camelCase TypeScript config into Firecracker's snake_case API payloads. Also validates configuration before boot.

```
import { VMConfig } from '@push.rocks/smartvm';

const vmConfig = new VMConfig({
  bootSource: { kernelImagePath: '/path/to/vmlinux' },
```

```
  machineConfig: { vcpuCount: 2, memSizeMib: 256 },
});

// Validate
const result = vmConfig.validate();
// { valid: true, errors: [] }

// Generate API payloads
vmConfig.toBootSourcePayload();
// { kernel_image_path: '/path/to/vmlinux' }

vmConfig.toMachineConfigPayload();
// { vcpu_count: 2, mem_size_mib: 256 }
```

`SocketClient` — Low-Level HTTP Client

Direct HTTP-over-Unix-socket communication with Firecracker. You typically don't need this directly — `MicroVM` handles it — but it's available if you want raw API access.

```
import { SocketClient } from '@push.rocks/smartvm';

const client = new SocketClient({ socketPath: '/tmp/firecracker.sock' });

const info = await client.get('/');
const putResult = await client.put('/machine-config', { vcpu_count: 2, mem_size_mib: 256 });
const patchResult = await client.patch('/vm', { state: 'Paused' });

// Check if socket is alive (polls with timeout)
const ready = await client.isReady(5000);
```

`SmartVMError` — Error Handling

All errors thrown by this module are `SmartVMError` instances with structured error codes.

```
import { SmartVMError } from '@push.rocks/smartvm';

try {
  await vm.start();
```

```
} catch (err) {
  if (err instanceof SmartVMError) {
    console.log(err.code);       // 'INVALID_CONFIG', 'SOCKET_TIMEOUT', 'API_ERROR', etc.
    console.log(err.statusCode); // HTTP status from Firecracker (if applicable)
    console.log(err.details);    // Raw error body from Firecracker (if applicable)
  }
}
```

Error codes:

Code	Description
`INVALID_STATE`	Operation not valid for current VM state
`INVALID_CONFIG`	Config validation failed
`SOCKET_TIMEOUT`	Firecracker socket didn't become ready
`API_TIMEOUT`	Firecracker API didn't respond in time
`SOCKET_REQUEST_FAILED`	HTTP request to socket failed
`API_ERROR`	Firecracker returned a non-2xx response
`BINARY_NOT_FOUND`	Firecracker binary not at expected path
`DOWNLOAD_FAILED`	Failed to download binary/kernel/rootfs
`VERSION_FETCH_FAILED`	Couldn't query GitHub for latest version
`BRIDGE_SETUP_FAILED`	Failed to create network bridge
`TAP_CREATE_FAILED`	Failed to create TAP device
`ROOTFS_CREATE_FAILED`	Failed to create blank rootfs
`ROOTFS_CLONE_FAILED`	Failed to clone rootfs image
`START_FAILED`	VM start sequence failed
`NO_CLIENT`	Socket client not initialized

Snapshots

Create and restore VM snapshots for fast cold-start or live migration:

```
// Pause first (required for snapshots)
await vm.pause();
```

```
// Create a snapshot
await vm.createSnapshot({
  snapshotPath: '/tmp/snapshot.bin',
  memFilePath: '/tmp/snapshot-mem.bin',
  snapshotType: 'Full',  // or 'Diff' for incremental
});

// Later: restore from snapshot
const freshVm = await smartvm.createVM({
  bootSource: { kernelImagePath: '/path/to/vmlinux' },
  machineConfig: { vcpuCount: 2, memSizeMib: 256 },
});
await freshVm.loadSnapshot({
  snapshotPath: '/tmp/snapshot.bin',
  memFilePath: '/tmp/snapshot-mem.bin',
  resumeVm: true,
});
```

MMDS (Metadata Service)

Pass metadata to your guest VM via Firecracker's Microvm Metadata Service:

```
const vm = await smartvm.createVM({
  bootSource: { /* ... */ },
  machineConfig: { /* ... */ },
  networkInterfaces: [{ ifaceId: 'eth0' }],
  mmds: {
    version: 'V2',
    networkInterfaces: ['eth0'],
  },
});

await vm.start();

// Set metadata from host
await vm.setMetadata({
  instance: { id: 'my-instance', region: 'eu-central-1' },
```

```
  secrets: { apiKey: 'sk-...' },
});

// Guest can access via: curl http://169.254.169.254/latest/meta-data/
const data = await vm.getMetadata();
```

Graceful Cleanup

The module registers cleanup handlers via `@push.rocks/smartexit` so resources are released even if your process crashes:

- □□ Firecracker child processes are killed
- □□ Unix socket files are removed
- □□ TAP devices are deleted
- □□ Bridge and NAT rules are torn down

You can also trigger cleanup manually:

```
// Stop one VM
await vm.stop();
await vm.cleanup();

// Stop all VMs and clean everything
await smartvm.stopAll();
await smartvm.cleanup();
```

TypeScript Interfaces

All configuration interfaces are fully exported for type-safe usage:

```
import type {
  ISmartVMOptions,
  IMicroVMConfig,
  IBootSource,
  IMachineConfig,
  IDriveConfig,
  INetworkInterfaceConfig,
```

```
  IVsockConfig,
  IBalloonConfig,
  IMmdsConfig,
  ILoggerConfig,
  IMetricsConfig,
  ISnapshotCreateParams,
  ISnapshotLoadParams,
  IRateLimiter,
  INetworkManagerOptions,
  ITapDevice,
  ISocketClientOptions,
  IApiResponse,
  TVMState,
  TFirecrackerArch,
  TCacheType,
  TSnapshotType,
  TLogLevel,
} from '@push.rocks/smartvm';
```

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 11:13:46 UTC by foss.global Team
Updated 2026-03-28 12:20:32 UTC by foss.global Team