Infrastructure as Code Engine: Design Document

Overview

This system parses declarative infrastructure configurations, computes the difference from the current state, builds a safe execution plan, and applies changes to cloud resources. The key architectural challenge is managing state, dependencies, and idempotent operations across distributed, eventually consistent cloud APIs.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This section provides foundational context for all milestones, setting the stage for the entire Infrastructure as Code engine.

Managing infrastructure manually—clicking through cloud consoles, running ad-hoc scripts, and keeping manual documentation—is a recipe for inconsistency, configuration drift, and operational risk. The modern approach is **Infrastructure as Code (IaC)**, treating servers, networks, and databases as declarative specifications that can be versioned, reviewed, and applied programmatically. This project is about building the engine that powers such a system: a tool that reads a desired state, compares it with reality, and safely orchestrates the necessary changes.

The core challenge lies in bridging the **declarative intent** of the user ("I want a web server with this configuration") with the **imperative actions** of cloud APIs ("create a VM, then attach a disk, then configure a firewall rule"), while maintaining a reliable record of what was actually built. This requires sophisticated state management, dependency analysis, and robust error handling in an environment of eventual consistency and rate-limited APIs.

Mental Model: The Master Blueprint and Site Manager

Imagine you are constructing a building. You start with a **master blueprint** (the IaC configuration)—a detailed, declarative document specifying every component: walls, electrical wiring, plumbing, and their exact properties. This blueprint is written in a specialized language (like HCL) that describes the *what*, not the *how*.

You have a **site manager** (the IaC engine) whose job is to make the real site match the blueprint. On their first day, they survey the empty lot (current state: nothing) and create a detailed **work order** (execution plan) listing every task: pour foundation, erect frame, install wiring. They must figure out the correct order—you can't install drywall before running electrical conduit (dependency resolution).

Once construction begins, the site manager keeps a meticulous **logbook** (state file) of what has been completed. For each item in the blueprint, they record the real-world identifier (e.g., "Wall Section A: constructed at coordinates X,Y,Z"). This logbook is crucial. When the blueprint is updated (e.g., "add a window to Wall Section A"), the manager consults the logbook to see what's already built, figures out the diff (add a window), and creates a new work order.

Crucially, the site manager uses **specialized subcontractors** (providers) for different tasks. An electrician (AWS provider) handles wiring, while a plumber (GCP provider) handles pipes. The manager communicates with each using their own terminology and processes (cloud APIs).

This mental model captures the essence of our IaC engine:

- **Blueprint** = Declarative configuration (`.tf` files)
- **Site Manager** = The core engine (Parser, Planner, Executor)
- **Logbook** = State file
- **Work Order** = Execution Plan
- **Subcontractors** = Providers (AWS, GCP, Azure plugins)

> The key insight is that the logbook (state) is the source of truth for the *real world*, while the blueprint (configuration) is the source of truth for the *desired world*. The engine's job is to reconcile the two safely and efficiently.

The Core Problem: Declarative to Imperative with State

Translating a static declaration of desired infrastructure into a dynamic set of API calls, while maintaining correctness and safety, involves several intertwined technical challenges:

1. **State Reconciliation and Idempotency:** The engine must know what currently exists to decide what needs to be created, updated, or destroyed. Simply re-running the blueprint from scratch every time is not feasible—it would destroy and recreate everything, causing downtime. The engine must perform a

three-way diff between the desired configuration, the last known state, and (in some cases) the actual live infrastructure. All operations must be **idempotent**: applying the same configuration multiple times should result in the same final state, regardless of intermediate failures or partial applies.

2. **Dependency and Ordering Management:** Infrastructure resources have complex relationships. A virtual machine depends on a network subnet, which depends on a virtual network. The engine must automatically infer these dependencies from the configuration (e.g., `subnet_id = aws_subnet.main.id` creates an implicit link) and allow explicit overrides (`depends_on`). It must then build a **Directed Acyclic Graph (DAG)** and execute changes in a correct order: creates in dependency order, updates with similar constraints, and destroys in reverse dependency order. Cycles in dependencies must be detected and rejected.
3. **Concurrency Control and Safety:** In a team setting, two engineers might try to apply different configurations to the same infrastructure simultaneously. This could lead to race conditions and corrupted state. The engine needs a **state locking mechanism** to ensure only one apply operation proceeds at a time. The locking must be robust to process crashes (avoiding stuck locks) and work across distributed environments (remote teams).
4. **Provider Abstraction and Heterogeneity:** Cloud APIs are diverse and ever-changing. The engine needs a clean **plugin architecture** that abstracts the specifics of AWS, Google Cloud, Azure, and even on-premise systems behind a uniform interface for Create, Read, Update, and Delete (CRUD) operations. Providers must handle authentication, rate limiting, retries with backoff, and the eventual consistency semantics of their respective clouds.
5. **Configuration Language and Expressivity:** Users need a language to declare resources, define variables, reuse modules, and output useful values. This language must be parsed, validated, and transformed into an internal representation. **Variable interpolation** (e.g., `${var.region}`) and **module composition** add layers of complexity, requiring multi-pass resolution and scope management.

The following table summarizes the core transformation the engine must perform at each stage:

Stage	Input	Core Challenge	Output
Parse & Resolve	HCL/YAML config files	Handling variables, modules, and complex expressions without infinite loops	A normalized, concrete graph of `Resource` objects with all references resolved.
State Diff	Desired Resources + Last Known State	Computing minimal, correct change set in the face of nested/computed attributes	A set of proposed `PlanAction` objects (Create, Update, Delete, No-op).
Plan	Desired Resources + State Diff + Dependencies	Ordering actions safely (no broken dependencies) and presenting a preview	An `ExecutionPlan` (DAG of `PlanAction` nodes) ready for application.
Apply	ExecutionPlan + Live Cloud APIs	Managing concurrency, retries, timeouts, and partial failures while preserving state consistency	Updated infrastructure and a new, accurate `StateFile`.

Decision: State as the Source of Truth

- **Context**: We need a reliable record of what the engine has provisioned to enable safe updates and deletes.
- **Options Considered**:
  1. **Pure Declarative (No State)**: Re-parse configuration and query all cloud APIs on every run to discover current state. This is simple but slow, expensive (API costs), and unreliable (APIs may not expose all attributes, transient resources may be missed).
  2. **State-First**: Maintain a persistent, versioned state file that records the resource IDs and attributes after each apply. The state is the primary input for computing diffs.
- **Decision**: **State-First**.
- **Rationale**: A state file provides a fast, deterministic, and complete record of managed resources. It allows the engine to work offline (planning) and reduces reliance on potentially flaky or rate-limited cloud APIs for baseline discovery. It's the established pattern used by Terraform and other mature tools, proven at scale.
- **Consequences**: We must now design robust state storage, locking, and serialization. The state file becomes a critical asset—corruption or loss can sever the link between configuration and real resources. We must also implement state refresh mechanisms to detect and reconcile "drift" (changes made outside the IaC tool).

Option	Pros	Cons	Chosen?
Pure Declarative	No state to manage or corrupt; always queries live infrastructure.	Very slow; API rate limits; cannot detect resources deleted via API; requires perfect API idempotency.	❌
State-First	Fast planning; clear audit trail; enables advanced features like move, taint, and targeted operations.	State file is a singleton bottleneck; requires locking; risk of state drift if not refreshed.	✅

Existing Approaches and Trade-offs

The IaC landscape offers several established tools, each embodying different design philosophies that inform our own architectural choices.

Decision: Terraform-like Declarative Model vs. Pulumi-like Imperative SDK

- **Context**: We must choose the core paradigm for how users define infrastructure.
- **Options Considered**:
  1. **Declarative DSL (Terraform)**: Users write configuration in a domain-specific language (HCL) or YAML/JSON. The engine is entirely responsible for planning and execution.
  2. **Imperative SDK (Pulumi, CDK)**: Users write full-fledged programs (Python, Go, TypeScript) that call SDK functions to declare resources. The program's execution *is* the planning phase.
- **Decision**: **Declarative DSL**.
- **Rationale**: For an educational engine focused on the core challenges of reconciliation and planning, a declarative model cleanly separates *intent* from *execution*. It forces us to build a sophisticated planner and state manager. An imperative SDK pushes more complexity into the user's code and the runtime, making the engine's job different (more about snapshotting and diffing program output). The declarative approach is also more accessible for beginners and aligns with the "blueprint" mental model.
- **Consequences**: We must build a parser and interpreter for a configuration language. Users lose the flexibility of loops and conditionals from general-purpose languages unless we replicate them in our DSL (e.g., `count` and `for_each`).

Option	Pros	Cons	Representative Tool
Declarative DSL	Clear separation of intent/execution; easy to analyze, validate, and plan; simple config files.	Limited expressiveness; custom language to learn; requires complex engine.	**Terraform**, CloudFormation, Ansible
Imperative SDK	Full power of a programming language; familiar syntax; easier abstraction and reuse.	Harder to reason about desired state; program execution required for planning; can be overkill.	**Pulumi**, AWS CDK, Crossplane

Terraform is the canonical model for our project. It uses a custom HCL, maintains a state file (`terraform.tfstate`), has a rich provider ecosystem, and performs a planning stage. Its architecture directly inspires our component breakdown: Parser (config load), State (backend), Graph (terraform graph), and Provider plugins. However, Terraform's codebase is large and complex. Our goal is to distill its core concepts into a manageable, educational implementation.

AWS CloudFormation takes a more centralized, cloud-vendor-controlled approach. Templates are JSON/YAML, and state is managed entirely by the AWS service. This simplifies the client but creates vendor lock-in and lacks the multi-cloud provider abstraction we aim to build. Its "change sets" are analogous to our execution plan.

Pulumi represents the "imperative SDK" approach. It allows infrastructure to be defined in real code, which is then evaluated to produce a desired state graph. Pulumi then uses a Terraform-like engine (often the Terraform engine itself via bridged providers) to plan and apply. For our project, adopting this model would mean focusing on capturing and diffing program snapshots rather than parsing a DSL.

> The trade-off is fundamentally about **control vs. flexibility**. A declarative engine (like Terraform) maintains tight control over the planning process, ensuring predictable and safe operations. An imperative SDK (like Pulumi) offers maximal flexibility to the user but places more trust in the user's code to correctly define the desired end state.

Our design will follow the **Terraform architectural pattern** but with significant simplifications for clarity and educational value:

- We will support a subset of HCL or a simpler YAML configuration for faster parsing.
- Our state management will start with a local file and a simple lock, later extending to remote backends.
- Our provider interface will be similar to Terraform's plugin system but less complex, focusing on the core CRUD lifecycle.

This approach gives us a well-trodden path to explore all the fascinating problems of IaC—parsing, state, graphs, and plugins—without getting lost in the intricacies of a full, production-grade language or SDK runtime.

> **Milestone(s):** This section sets the scope for the entire project and is relevant to all milestones.

Goals and Non-Goals

This section clearly defines the project's boundaries, separating what the IaC engine will accomplish from what it deliberately omits. A well-defined scope is critical for managing complexity and ensuring the core value proposition — a reliable, understandable, and extensible engine for managing infrastructure declaratively — is delivered successfully.

> **Architectural Principle:** A system's boundaries are defined as much by what it chooses *not* to do as by what it does. Explicit non-goals prevent scope creep and allow focused effort on the core, differentiating capabilities.

Goals

The primary goal is to build a functional, educational IaC engine that demonstrates the fundamental patterns and challenges of reconciling desired state with actual state in a cloud environment. The following table enumerates the specific capabilities the system must deliver.

Goal	Description	Rationale & Key Capabilities
Declarative Configuration	Accept user-defined configuration files that describe the *desired* end-state of infrastructure, not the steps to create it.	This is the foundational philosophy of IaC. The engine must parse a configuration DSL (HCL-like syntax or YAML), resolve variables and modules, and produce a normalized set of `Resource` objects representing the intended infrastructure.
State Management	Maintain a persistent, mutable record of the *actual* state of deployed infrastructure, enabling idempotent operations and change detection.	State is the engine's memory. Without it, every `apply` would be a blind creation. The system must store attribute values and resource IDs, support safe concurrent access via locking, and reliably compute the difference between desired and actual state.
Dependency-Aware Planning	Analyze resource relationships to build an execution order (a Directed Acyclic Graph) and generate a safe, previewable plan of changes.	Infrastructure resources depend on each other (e.g., a subnet must exist before a VM placed in it). The planner must infer these dependencies, detect cycles, topologically sort resources, and produce a sequence of `PlanAction` items (CREATE, UPDATE, DELETE, NOOP) that can be reviewed before execution.
Provider Plugin Abstraction	Define a clean interface (`BaseProvider`) that abstracts the specific CRUD operations of various cloud platforms (AWS, GCP, Azure) or services.	The engine cannot hardcode support for every cloud API. A plugin architecture allows the core logic to remain stable while providers for new services are developed independently. Each provider implements the lifecycle operations for its resource types.
Safe, Idempotent Apply	Execute the plan against real infrastructure in a controlled manner, handling API failures, retries, and ensuring operations are idempotent where possible.	The `Executor` must coordinate with providers, manage concurrency, implement retry logic with backoff, and update the state file only upon successful completion of operations. This ensures the system is robust against transient failures and avoids duplicate resources.
Educational Codebase	Produce a codebase that is readable, well-structured, and illustrative of distributed systems and compiler design patterns.	The implementation should prioritize clarity over excessive optimization. The architecture should be modular, with clear interfaces and data flow, making it a valuable learning tool for understanding how tools like Terraform work under the hood.

Non-Goals

Equally important are the features and responsibilities the system will *not* undertake. These exclusions keep the project focused and manageable, often delegating complexity to external systems or defining them as future extensions.

Non-Goal	Why It's Out of Scope	Recommended Alternative / Note
A Complete, Production-Grade HCL Parser	Implementing the full HCL 2.x syntax with all expressions, functions, and blocks is a massive undertaking distinct from the core IaC reconciliation engine.	Use a simplified, HCL-*like* syntax or YAML as the configuration language. For a more realistic parser, leverage an existing library (e.g., Python's `pyhcl` or Go's `hcl` package) in the **Implementation Guidance**, but the core design document will describe a simpler, custom parser for educational clarity.
Built-in Providers for All Cloud Services	The engine is a platform, not a comprehensive cloud management tool. Building and maintaining providers for hundreds of resource types across multiple clouds is a separate, vast project.	The design includes a provider SDK (`BaseProvider` interface). The implementation will include one or two sample providers (e.g., a mock provider and a simple AWS EC2 provider) to demonstrate the pattern. Support for additional providers is left as an exercise or extension.
Web-Based UI or Dashboard	The primary interface is the command line. Building a graphical interface is a significant front-end project that does not contribute to the core learning objectives of state management, planning, and provider abstraction.	The CLI provides all necessary functionality (`plan` , `apply`). A UI could be built as a separate, consuming application that calls the engine's core libraries.
Continuous Drift Detection & Remediation	Automatically detecting and correcting configuration drift on a schedule is an operational feature that builds upon the core `plan` / `apply` mechanics. It introduces scheduling, eventing, and policy decision complexity.	Drift detection can be simulated by running `plan` periodically (e.g., via cron). Automatic remediation would require an automated `apply` , which is a security/policy decision beyond the engine's responsibility.
Advanced Policy-as-Code (e.g., OPA/Sentinel)	Integrating policy checks that validate plans against security, compliance, or cost rules is a critical enterprise feature but adds significant complexity in policy language, evaluation engine, and enforcement hooks.	The `plan` output can be piped to external policy tools. The architecture could be extended later with a validation webhook between planning and execution.
Team Collaboration Features (RBAC, Audit Logging)	Managing user permissions, audit trails, and approval workflows is essential for team use but belongs in a surrounding CI/CD or orchestration layer, not the core state reconciliation engine.	Use the engine with a CI/CD system (like the one built in the prerequisite project) which handles authentication, authorization, and logging of who ran which plans.
Multi-Cloud Dependency Resolution	The engine does not need to handle dependencies *between* different cloud providers (e.g., an AWS resource depending on a GCP resource). This is an extreme edge case that vastly complicates authentication and state storage.	Assume a single cloud provider context per configuration/state file. Cross-provider setups can be managed by using the engine separately for each provider and passing outputs manually or via a wrapper script.
Native Support for Imperative Scripts	The engine is declarative. While it may need to execute provisioning scripts *as a resource action* (e.g., via a `local_exec` provider), it will not parse or execute general-purpose imperative orchestration scripts as its primary config.	Use a declarative resource provided by a custom provider to wrap imperative scripts if absolutely necessary, acknowledging that this breaks idempotency guarantees.

> **Key Insight:** The distinction between goals and non-goals is not about importance, but about layering. The engine aims to be an excellent *platform* for declarative infrastructure reconciliation. Features like UI, collaboration, and policy are best built *on top* of this platform, not baked into its core.

High-Level Architecture

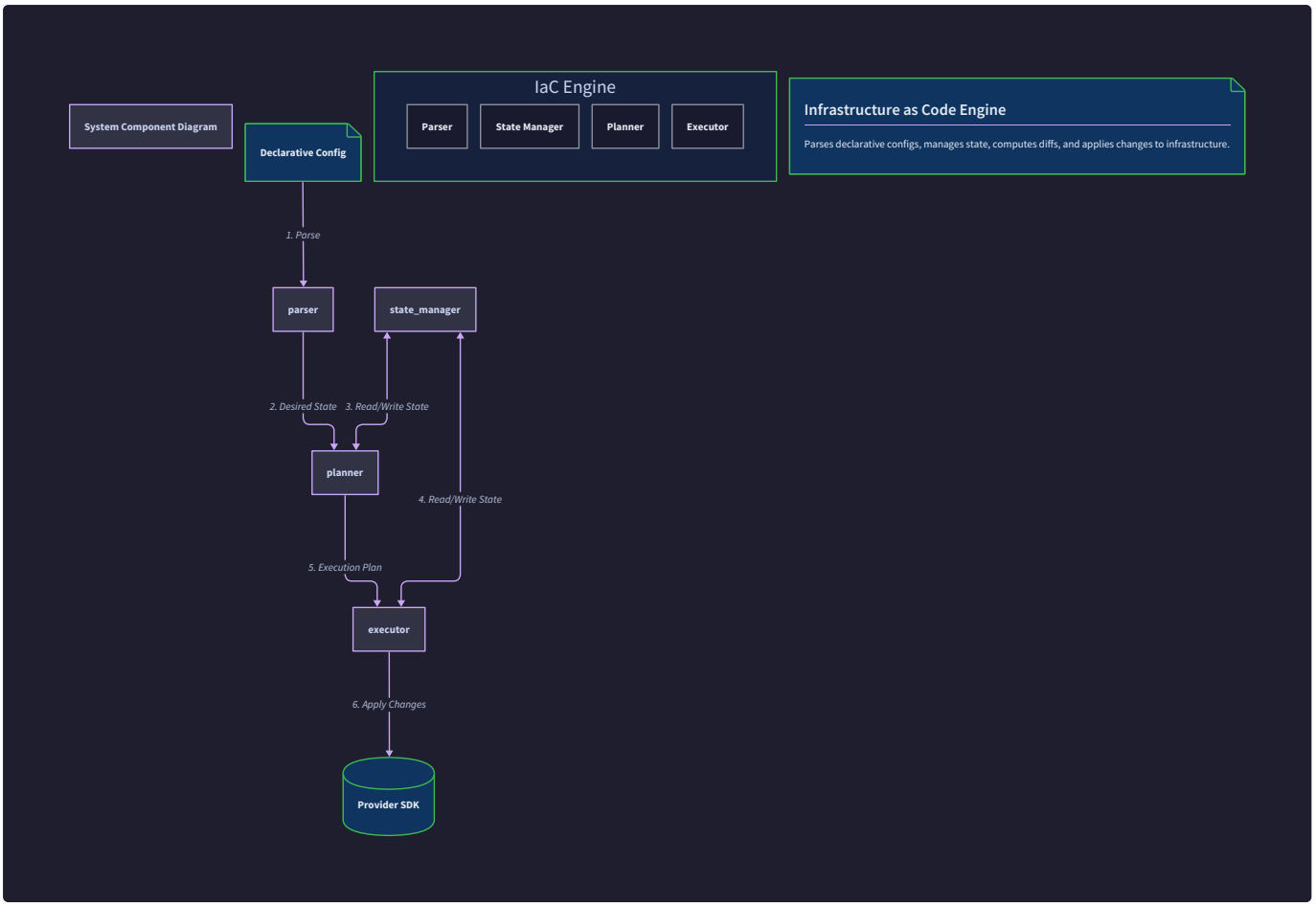
> **Milestone(s):** This section provides a system-wide overview and is relevant to all milestones (1, 2, 3, 4).

This section presents the bird's-eye view of the Infrastructure as Code (IaC) engine. Before diving into individual component details, it's crucial to understand how the system fits together as a cohesive whole. We'll first introduce the five core components and their responsibilities, then describe the recommended project structure to organize this architecture in code.

Component Overview and Responsibilities

Think of the IaC engine as a **construction project management office**. A project begins with architectural blueprints (configuration files). The office has specialists who interpret these blueprints, check what's already built, figure out the build order, and finally coordinate with various construction crews (cloud APIs) to execute the work. Our system decomposes this workflow into five specialized components.

Each component has a single, well-defined responsibility and communicates with others through clear interfaces. This separation of concerns allows for independent development, testing, and potential replacement of individual parts (e.g., swapping the parser from HCL to YAML).



The five core components are:

1. **Configuration Parser**: The **Interpreter and Expander**. This component consumes human-written declarative configuration files (HCL or YAML). Its job is to read the raw text, resolve all variables and module references, and produce a complete, normalized, in-memory representation of the desired infrastructure—a collection of `Resource` objects with concrete attribute values. It handles the complexity of the Domain-Specific Language (DSL) so the rest of the system can work with simple data structures.
2. **State Manager**: The **Ledger and Lock**. This component is the system's memory. It persistently stores the last known state of the deployed infrastructure in a **state file**. More importantly, it manages concurrent access to this state via a **lock**, preventing two simultaneous `apply` operations from corrupting each other's work. It can also compute the difference (diff) between the desired state (from the Parser) and the current state (from the ledger), which is the foundation of planning.
3. **Planner (Dependency Graph & Planning)**: The **Project Manager and Gantt Chart**. This component takes the normalized resources from the Parser and the current state from the State Manager. It analyzes the resources to build a **Directed Acyclic Graph (DAG)** based on their dependencies (e.g., a subnet must exist before a virtual machine placed within it). Using this graph and the state diff, it generates a safe, ordered **execution plan**—a list of `PlanAction` objects specifying precisely what needs to be created, updated, deleted, or left unchanged. This plan can be previewed by the user before any changes are made.
4. **Provider Abstraction (SDK)**: The **Universal Remote Control Interface**. This is not a single component but an abstraction layer. It defines a standard `BaseProvider` interface with **CRUD** operations (`create` , `read` , `update` , `delete`). Concrete implementations (like `AwsProvider` , `GoogleCloudProvider`) act as plugins that translate these generic calls into specific API calls for their respective cloud platforms. This design allows the engine to support multiple clouds without changing its core logic.
5. **Executor**: The **Foreman and Dispatcher**. This component takes the validated execution plan from the Planner and makes it happen. It is responsible for carrying out the plan by calling the appropriate methods on the configured `BaseProvider` implementations. It manages the execution flow: respecting the dependency order, handling concurrency for independent resources, implementing retry logic with exponential backoff for transient API failures, and reporting progress and outcomes back to the State Manager to update the ledger.

The table below summarizes each component's key responsibility, its primary inputs, and its primary outputs.

Component	Primary Responsibility	Key Inputs	Key Outputs
Configuration Parser	Transforms declarative config files into a normalized set of concrete resources.	HCL/YAML configuration files, variable values, module sources.	A collection of resolved `Resource` objects.
State Manager	Persists the known state of infrastructure and manages safe, concurrent access to it.	Desired `Resource` collection, current state file, lock request.	`StateRecord` collection, state diff, lock token.
Planner	Analyzes dependencies and state differences to produce a safe, ordered execution plan.	Desired `Resource` collection, current `StateRecord` collection.	`DependencyGraph`, ordered list of `PlanAction` objects (the plan).
Provider SDK	Defines the interface for cloud-specific operations and provides plugin implementations.	`PlanAction` (via Executor), provider configuration (credentials, region).	Real cloud resource ID and attributes (on success) or error.
Executor	Orchestrates the execution of a plan by calling provider methods and managing state updates.	Ordered `PlanAction` list, configured `BaseProvider` instances.	Updated `StateRecord` collection (written back via State Manager), execution result summary.

Data Flow and Interactions: The components interact in a specific sequence during the two primary workflows: `plan` and `apply`. During a `plan`, data flows from left to right in the diagram: Config → Parser → Planner. The Planner also reads from the State Manager. The final plan is output to the user. During an `apply`, the loop is closed: the Executor takes the plan, interacts with Providers to change the real infrastructure, and then writes the new state back through the State Manager. The State Manager's lock is acquired at the start of `apply` and released at the end, ensuring isolation.

Recommended File and Module Structure

A well-organized codebase mirrors the architectural separation. The following structure is recommended for the Python implementation. It uses a clear separation between the core engine, provider plugins, and command-line interface, following Python packaging best practices.

```
infra-as-code-engine/
├── pyproject.toml                    # Project metadata and dependencies
├── README.md
├── src/
│   └── iac_engine/                   # Main package
│       ├── __init__.py
│       ├── cli.py                    # Command-line interface (plan, apply, destroy)
│       │
│       ├── core/                     # The core engine components
│       │   ├── __init__.py
│       │   ├── parser.py             # Configuration Parser component
│       │   ├── state_manager.py      # State Manager component
│       │   ├── planner.py            # Planner component
│       │   ├── executor.py           # Executor component
│       │   └── models.py             # Core data types: Resource, StateRecord, etc.
│       │
│       ├── providers/                # Provider Abstraction SDK and plugins
│       │   ├── __init__.py
│       │   ├── base.py               # BaseProvider abstract class
│       │   ├── aws/                  # AWS provider implementation
│       │   │   ├── __init__.py
│       │   │   ├── provider.py       # AwsProvider class
│       │   │   └── resources/        # AWS-specific resource implementations
│       │   │       ├── ec2_instance.py
│       │   │       └── s3_bucket.py
│       │   ├── google/               # Google Cloud provider
│       │   └── mock/                 # Mock provider for testing
│       │
│       └── utils/                    # Shared utilities
│           ├── __init__.py
│           ├── file_lock.py          # Locking implementation
│           ├── retry.py              # Retry with exponential backoff
│           └── graph.py              # Generic DAG utilities
│
├── tests/                            # Comprehensive test suite
│   ├── unit/
│   │   ├── test_parser.py
│   │   ├── test_state_manager.py
│   │   └── ...
│   ├── integration/
│   └── e2e/
│
├── examples/                         # Example configuration files
│   ├── simple-webapp/
│   │   ├── main.hcl
│   │   └── variables.yaml
│   └── multi-cloud/
│
└── docs/                             # Design docs, ADRs
    └── architecture_decisions.md
```

Key Rationale for This Structure:

- **`src/` layout**: Using a `src` directory helps avoid common issues with Python imports, ensuring the package is always imported correctly, whether in development or after installation.
- **Clear namespace**: The main package `iac_engine` provides a clear namespace for all engine-related code.
- **Component isolation**: Each core component (`parser.py` , `state_manager.py` , etc.) lives in the `core/` module, making dependencies between them explicit and preventing circular imports.
- **Plugin architecture**: The `providers/` directory is structured to easily add new providers as sub-packages. The `base.py` defines the contract that all plugins must follow.
- **Shared utilities**: Common patterns like locking, retries, and graph algorithms are factored out into `utils/` to avoid duplication and ensure consistency.
- **Separation of tests**: Tests mirror the source structure, making it easy to locate tests for a specific module.

This structure scales well. As the project grows—adding more providers, utility functions, or complex examples—new directories and files can be added within the existing logical framework without major restructuring.

Implementation Guidance

This subsection provides concrete, actionable guidance for implementing the high-level architecture described above in Python.

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Production Readiness)
Configuration Parsing	Use `pyhcl2` (HCL2 parser) or `pyyaml` for YAML. Manual variable interpolation.	Implement a full recursive-descent parser for a custom DSL with advanced expression evaluation.
State Serialization	JSON files for local state. Simple file-based locking.	Protocol Buffers or MessagePack for compact binary format. Distributed lock via DynamoDB or Consul.
Dependency Graph	Python's `networkx` library for graph operations, or a custom `dict`-based adjacency list.	Custom incremental graph with persistent indexing for fast dependency queries on large state.
Provider SDK	Abstract Base Classes (`abc.ABC`) for interface definition. `boto3` for AWS, `google-cloud-*` libraries for GCP.	gRPC service definitions for a plugin system where providers run as separate processes.
Concurrency & Retries	`concurrent.futures.ThreadPoolExecutor` for parallel applies. Custom retry decorator.	`asyncio` with `aiohttp` for async I/O. Circuit breaker pattern (e.g., `pybreaker`) for API failure handling.

B. Recommended File and Module Structure (Detailed)

The following Python code establishes the foundational modules and data models. Create these files in the `src/iac_engine/` directory as shown in the structure above.

1. Core Data Models (`src/iac_engine/core/models.py`):

PYTHON

```python
"""Core data models for the IaC engine."""

from enum import Enum

from typing import Any, Dict, List, Optional

from dataclasses import dataclass, field


class ActionType(Enum):

    """The type of change to be performed on a resource."""

    CREATE = "create"

    UPDATE = "update"

    DELETE = "delete"

    NOOP = "no-op"  # No operation needed


@dataclass

class Resource:

    """A desired infrastructure resource as defined in configuration."""

    id: str  # Unique identifier within the config (e.g., "aws_instance.web")

    type: str  # Provider resource type (e.g., "aws_instance")

    name: str  # Local name (e.g., "web")

    attributes: Dict[str, Any] = field(default_factory=dict)  # Configuration attributes


@dataclass

class StateRecord:

    """A record of a resource as it exists in the deployed infrastructure."""

    resource_id: str  # Matches Resource.id

    resource_type: str

    resource_name: str

    attributes: Dict[str, Any] = field(default_factory=dict)  # Last known attributes

    dependencies: List[str] = field(default_factory=list)  # IDs of resources this depends on


@dataclass

class PlanAction:

    """A proposed change to bring reality in line with configuration."""

    action_type: ActionType

    resource: Resource  # The desired resource state

    prior_state: Optional[StateRecord] = None  # The state before the action (for UPDATE/DELETE)

    new_state: Optional[StateRecord] = None  # The expected state after the action


@dataclass

class DependencyGraphNode:

    """A node in the resource dependency graph."""

    resource_id: str
```

```
    depends_on: List[str] = field(default_factory=list)  # Outgoing edges

    required_by: List[str] = field(default_factory=list)  # Incoming edges (for efficient traversal)
```

2. Provider Base Interface (`src/iac_engine/providers/base.py`):

PYTHON

```python
"""Abstract base class for all infrastructure providers."""

from abc import ABC, abstractmethod

from typing import Any, Dict, Optional

from ..core.models import Resource, StateRecord


class BaseProvider(ABC):

    """Interface that all concrete providers must implement."""



    @abstractmethod

    def create(self, resource: Resource) -> StateRecord:

        """Create a new resource.



        Args:

            resource: The desired resource configuration.



        Returns:

            A StateRecord representing the newly created resource, including its

            server-assigned ID and final attributes.



        Raises:

            ProviderError: If creation fails.

        """

        pass



    @abstractmethod

    def read(self, resource_id: str, resource_type: str) -> Optional[StateRecord]:

        """Read the current state of a resource from the provider.



        Args:

            resource_id: The unique identifier of the resource.

            resource_type: The type of resource.



        Returns:

            A StateRecord if the resource exists, None if it does not.



        Raises:

            ProviderError: If the read operation fails (e.g., network error).

        """

        pass
```

```
    @abstractmethod
    def update(self, resource: Resource, prior_state: StateRecord) -> StateRecord:
        """Update an existing resource.

        Args:
            resource: The new desired configuration.
            prior_state: The state of the resource before the update.

        Returns:
            A StateRecord representing the updated resource.

        Raises:
            ProviderError: If update fails or the resource cannot be updated in-place.
        """
        pass

    @abstractmethod
    def delete(self, resource_id: str, resource_type: str, prior_state: StateRecord) -> None:
        """Delete an existing resource.

        Args:
            resource_id: The unique identifier of the resource.
            resource_type: The type of resource.
            prior_state: The last known state of the resource.

        Raises:
            ProviderError: If deletion fails.
        """
        pass

    @abstractmethod
    def validate_credentials(self) -> bool:
        """Validate that the provider is properly configured and credentials work.

        Returns:
            True if credentials are valid, False otherwise.
        """
```

```
        pass
```

C. Infrastructure Starter Code

1. Atomic File Operations Utility (`src/iac_engine/utils/atomic_file.py`):

PYTHON

```python
"""Utilities for atomic file writes to prevent corruption."""

import os

import tempfile

import shutil

from pathlib import Path

from typing import Any

import json


def write_atomic_json(filepath: Path, data: Any) -> None:

    """Write JSON data to a file atomically.


    Writes to a temporary file in the same directory, then renames it

    to the target filepath. This prevents partial writes if the process

    crashes during write.


    Args:

        filepath: The destination file path.

        data: Serializable data to write as JSON.

    """

    filepath.parent.mkdir(parents=True, exist_ok=True)


    # Create a temporary file in the same directory

    with tempfile.NamedTemporaryFile(

        mode='w',

        dir=filepath.parent,

        prefix=f".{filepath.name}.tmp.",

        delete=False

    ) as tmp_file:

        json.dump(data, tmp_file, indent=2)

        tmp_path = Path(tmp_file.name)


    try:

        # Atomic rename (POSIX guarantees this is atomic)

        tmp_path.rename(filepath)

    except Exception:

        # If rename fails, clean up the temporary file

        tmp_path.unlink(missing_ok=True)

        raise


def read_json_with_backup(filepath: Path) -> Any:
```

```
    """Read JSON file, with automatic fallback to a backup if present.

    Args:
        filepath: The primary file path to read.

    Returns:
        The parsed JSON data.

    Raises:
        FileNotFoundError: If neither the file nor its backup exists.
        json.JSONDecodeError: If the file contains invalid JSON.
    """
    backup_path = filepath.with_suffix(filepath.suffix + ".backup")

    for path in [filepath, backup_path]:
        if path.exists():
            try:
                with open(path, 'r') as f:
                    return json.load(f)
            except json.JSONDecodeError:
                # If the main file is corrupt, try the backup
                if path == filepath and backup_path.exists():
                    continue
                raise

    raise FileNotFoundError(f"State file not found: {filepath}")
```

D. Core Logic Skeleton Code

1. Main CLI Entry Point (`src/iac_engine/cli.py`):

PYTHON

```python
"""Command-line interface for the IaC engine."""

import argparse

from pathlib import Path

from typing import Optional


# TODO: Import the core components once implemented

# from .core.parser import ConfigurationParser

# from .core.state_manager import StateManager

# from .core.planner import Planner

# from .core.executor import Executor


def plan_command(config_path: Path, state_path: Path, var_file: Optional[Path]) -> None:

    """Generate and show an execution plan.


    Args:

        config_path: Path to the main configuration file.

        state_path: Path to the state file.

        var_file: Optional path to a file containing variable values.

    """

    print(f"Planning for config: {config_path}")


    # TODO 1: Initialize the ConfigurationParser and parse the config file

    # parser = ConfigurationParser()

    # desired_resources = parser.parse_file(config_path, var_file)


    # TODO 2: Initialize the StateManager and read the current state

    # state_manager = StateManager(state_path)

    # current_state = state_manager.read_state()


    # TODO 3: Initialize the Planner, build the dependency graph, and generate plan

    # planner = Planner()

    # execution_plan = planner.generate_plan(desired_resources, current_state)


    # TODO 4: Display the plan in a human-readable format

    # print(execution_plan.summary())


    print("Plan command not yet implemented.")


def apply_command(config_path: Path, state_path: Path, var_file: Optional[Path], auto_approve: bool) -> None:

    """Apply the changes required to reach the desired state.
```

```
    Args:
        config_path: Path to the main configuration file.
        state_path: Path to the state file.
        var_file: Optional path to a file containing variable values.
        auto_approve: If True, skip interactive approval.
    """
    print(f"Applying config: {config_path}")
    # TODO: Implement apply workflow
    print("Apply command not yet implemented.")

def main():
    """Parse command line arguments and dispatch to appropriate function."""
    parser = argparse.ArgumentParser(description="Infrastructure as Code Engine")
    subparsers = parser.add_subparsers(dest='command', required=True)

    # Plan command
    plan_parser = subparsers.add_parser('plan', help='Generate an execution plan')
    plan_parser.add_argument('config', type=Path, help='Path to configuration file')
    plan_parser.add_argument('--state', type=Path, default=Path("./terraform.tfstate"), help='Path to state file')
    plan_parser.add_argument('--var-file', type=Path, help='Path to variable definitions file')

    # Apply command
    apply_parser = subparsers.add_parser('apply', help='Apply the execution plan')
    apply_parser.add_argument('config', type=Path, help='Path to configuration file')
    apply_parser.add_argument('--state', type=Path, default=Path("./terraform.tfstate"), help='Path to state file')
    apply_parser.add_argument('--var-file', type=Path, help='Path to variable definitions file')
    apply_parser.add_argument('--auto-approve', action='store_true', help='Skip interactive approval')

    args = parser.parse_args()

    if args.command == 'plan':
        plan_command(args.config, args.state, args.var_file)
    elif args.command == 'apply':
        apply_command(args.config, args.state, args.var_file, args.auto_approve)

if __name__ == "__main__":
    main()
```

E. Language-Specific Hints

- **Data Classes:** Use `@dataclass` from the `dataclasses` module (as shown) for the core models. They provide a clean, boilerplate-free way to define data structures with type hints.
- **Path Management:** Use `pathlib.Path` instead of string paths for all file operations. It provides a more object-oriented and cross-platform API.
- **Type Hints:** Make extensive use of Python type hints (`from typing import ...`). They serve as documentation and enable static type checking with tools like `mypy`, catching many errors early.
- **Abstract Base Classes:** The `abc` module is perfect for defining the `BaseProvider` interface. Use `@abstractmethod` to enforce implementation in subclasses.
- **Serialization:** For the state file, `json` is simple and human-readable. Use `json.dump` with `indent=2` for debugging. For production, consider adding a schema validation step before writing.

F. Milestone Checkpoint

After setting up the project structure and the skeleton code above, you should be able to:

1. Create a virtual environment and install basic dependencies (e.g., `pip install pyyaml`).
2. Run the CLI with `python -m src.iac_engine.cli plan ./examples/simple-webapp/main.hcl` and see the "not yet implemented" message.
3. Verify that the core data models can be imported and instantiated correctly by writing a simple test script:

```python
from iac_engine.core.models import Resource, ActionType

r = Resource(id="test.vm", type="mock_instance", name="vm", attributes={"size": "large"})

print(r)  # Should print a readable representation
```

This confirms the foundational project setup is correct before diving into component-specific implementation in the following milestones.

Data Model

> **Milestone(s):** This section defines the foundational data structures used throughout the entire IaC engine and is relevant to all milestones (1, 2, 3, 4).

The **Data Model** serves as the common language spoken by all components of the IaC engine. It defines the core entities—resources, state records, plan actions, and dependency nodes—that flow through the system as infrastructure configurations are transformed from declarative intent into actual deployed resources. A consistent, well-defined data model is crucial because it ensures that the parser, state manager, planner, and executor can communicate without ambiguity, even as each component performs vastly different operations. This section provides the complete specification for these data structures and maps their lifecycle transformations.

Mental Model: The Recipe, the Pantry Inventory, the Shopping List, and the Cooking Steps

Imagine you're preparing a complex meal:

1. **The Recipe (`Resource`):** This is your desired end state—a list of ingredients and instructions. "2 tomatoes, 1 onion, sauté for 10 minutes." It's declarative: you specify *what* you want, not the imperative steps to get it if ingredients are missing.
2. **The Pantry Inventory (`StateRecord`):** This is a snapshot of what you currently have. "Pantry contains 1 tomato, 0 onions." It's a record of reality at a specific point in time.
3. **The Shopping List (`PlanAction`):** By comparing the recipe to your pantry, you generate a list of actions needed: "BUY 1 tomato, BUY 1 onion." This is the **execution plan**—a set of imperative changes to bridge the gap between desired state (recipe) and current state (pantry).
4. **The Cooking Dependency Chart (`DependencyGraphNode`):** Some steps depend on others. You must chop the onion (Resource A) before you can sauté it with the tomatoes (Resource B). This chart defines the order in which actions must be executed.

In our IaC engine, the **Parser** reads the recipe (HCL/YAML config). The **State Manager** maintains the pantry inventory (state file). The **Planner** compares the two to create the shopping list (execution plan) and consults the dependency chart to put the list in the correct order. Finally, the **Executor** goes to the store and cooks (applies the plan via providers).

Key Types and Structures

The following tables define the primary data structures. All components must use these exact definitions to ensure interoperability. The fields are designed to carry all necessary information through each stage of the pipeline, from configuration parsing to final state persistence.

`Resource`: The Declarative Intent

A `Resource` represents a single infrastructure object as declared in the configuration file. It is the desired state specification before any runtime or state information is attached.

Field Name	Type	Description
`id`	`string`	**A unique, immutable identifier for the resource *within the configuration*.** This is typically constructed by the parser as `{resource_type}.{resource_name}` (e.g., `aws_instance.web_server`). It is used as the primary key for referencing resources in dependencies and within the state file. It is distinct from a cloud provider's runtime ID (e.g., `i-0abc123def456`).
`type`	`string`	**The type of cloud resource.** This maps directly to a provider's resource type. Examples: `aws_instance`, `google_compute_instance`, `kubernetes_deployment`. The provider interface uses this field to route CRUD operations to the correct implementation.
`name`	`string`	**The logical name given to the resource by the user in the configuration.** This is the local identifier within a configuration file or module. Example: In `resource "aws_instance" "web_server" { ... }`, the `name` is `"web_server"`.
`attributes`	`map[string]any`	**The complete set of configuration properties for the resource.** This is a key-value map where keys are attribute names (e.g., `ami`, `instance_type`, `tags`) and values are the fully resolved, interpolated values after variable and module processing. The structure of this map must conform to the schema expected by the corresponding provider.

Example: A resource block `resource "aws_s3_bucket" "data_lake" { bucket = "my-unique-name" }` would be parsed into a `Resource` object with:

- `id`: `"aws_s3_bucket.data_lake"`
- `type`: `"aws_s3_bucket"`
- `name`: `"data_lake"`
- `attributes`: `{"bucket": "my-unique-name", "acl": null, ...}` (including default or omitted attributes).

`StateRecord`: The Known Reality

A `StateRecord` is a snapshot of an infrastructure resource as it exists (or is believed to exist) in the real world. It is persisted to the **state file** and serves as the system's "memory" of what it has previously created or managed.

Field Name	Type	Description
`resource_id`	`string`	**The unique identifier of the resource, matching the `Resource.id` from configuration.** This creates the link between a configuration declaration and its recorded state.
`resource_type`	`string`	**The type of the resource.** Duplicated from the corresponding `Resource.type` for convenience and integrity checks.
`resource_name`	`string`	**The logical name of the resource.** Duplicated from the corresponding `Resource.name`.
`attributes`	`map[string]any`	**The actual attribute values as reported by the cloud provider's API during the last successful `read` or `apply` operation.** This includes **all** provider-output attributes, not just those set in configuration (e.g., a VM's `public_ip` assigned by the cloud, or a database's `arn`). This map is the source of truth for calculating diffs.
`dependencies`	`list[string]`	**A list of `resource_id` values that this resource depends on.** This is persisted to avoid the need to re-compute the dependency graph from configuration on every run, which is critical for operations like `destroy` where the configuration may no longer be available. It captures both explicit (`depends_on`) and implicit (attribute reference) dependencies.

Example: After creating the S3 bucket above, a `StateRecord` might be:

- `resource_id`: `"aws_s3_bucket.data_lake"`
- `resource_type`: `"aws_s3_bucket"`
- `resource_name`: `"data_lake"`
- `attributes`: `{"bucket": "my-unique-name", "arn": "arn:aws:s3:::my-unique-name", "region": "us-east-1", ...}`
- `dependencies`: `[]`

`PlanAction`: The Proposed Change

A `PlanAction` represents a single discrete change that needs to be applied to the infrastructure to align reality with the desired state. An **execution plan** is an ordered list of `PlanAction` objects.

Field Name	Type	Description
`action_type`	`enum(ActionType)`	**The type of change to perform.** The possible values are: - `ActionType.CREATE` : The resource does not exist in state and must be created. - `ActionType.UPDATE` : The resource exists, but one or more of its `attributes` have changed. - `ActionType.DELETE` : The resource exists in state but is no longer present in the configuration. - `ActionType.NOOP` : The resource exists and its configuration matches the state; no action is required.
`resource`	`Resource`	**The desired state of the resource after the action completes.** For `CREATE` and `UPDATE` , this is the target configuration. For `DELETE` , this field contains the resource's last known configuration (useful for provider operations). For `NOOP` , it's the current configuration.
`prior_state`	`StateRecord` (optional)	**The state of the resource before the action, if it exists.** This is `null` for `CREATE` actions. For `UPDATE` and `DELETE` , it holds the `StateRecord` read from the state file. It is used by the provider's `update` method to compute a partial update and by the executor for rollback information.
`new_state`	`StateRecord` (optional)	**The expected state of the resource after the action succeeds.** This is populated by the **Planner** as a *prediction* and is later updated by the **Executor** with the *actual* results from the provider. It starts with the `attributes` from `resource` and will be merged with provider-output attributes after a successful `create` or `update` .

Example: If you change the S3 bucket's `acl` attribute from `private` to `public-read` , the planner generates a `PlanAction` with:

- `action_type` : `ActionType.UPDATE`
- `resource` : `Resource{id: "aws_s3_bucket.data_lake", attributes: {"bucket": "my-unique-name", "acl": "public-read", ...}}`
- `prior_state` : `StateRecord{...attributes: {"acl": "private", ...}}`
- `new_state` : `StateRecord{...attributes: {"acl": "public-read", ...}}` (predicted)

`DependencyGraphNode` : The Ordering Constraint

A `DependencyGraphNode` is a vertex in the **Directed Acyclic Graph (DAG)** that encodes the order dependencies between resources. The graph is built by the **Planner** and is used to topologically sort the `PlanAction` list.

Field Name	Type	Description
`resource_id`	`string`	**The identifier of the resource this node represents.** Matches `Resource.id` and `StateRecord.resource_id` . This is the node's unique key in the graph.
`depends_on`	`list[string]`	**List of `resource_id` values that *this* resource depends on.** These are the outgoing edges. For example, if a subnet must exist before a VM can be created in it, the VM's node will have the subnet's `resource_id` in its `depends_on` list.
`required_by`	`list[string]`	**List of `resource_id` values that depend on *this* resource.** These are the incoming edges. This field is often derived for convenience during graph traversal and topological sorting (it is the inverse of `depends_on` relationships across the graph).

Example: A network interface (`aws_network_interface.main`) depends on a subnet (`aws_subnet.main`). Their graph nodes would be:

- Subnet Node: `resource_id: "aws_subnet.main", depends_on: [], required_by: ["aws_network_interface.main"]`
- Interface Node: `resource_id: "aws_network_interface.main", depends_on: ["aws_subnet.main"], required_by: []`

Supporting Enum: `ActionType`

This enumeration defines the four fundamental operations in a plan.

Constant	Description
`ActionType.CREATE`	The resource must be created. Corresponds to the provider's `create` method.
`ActionType.UPDATE`	The resource must be updated in-place. Corresponds to the provider's `update` method.
`ActionType.DELETE`	The resource must be destroyed. Corresponds to the provider's `delete` method.
`ActionType.NOOP`	No operation is needed. The resource is already in sync.

Type Relationships and Lifecycle

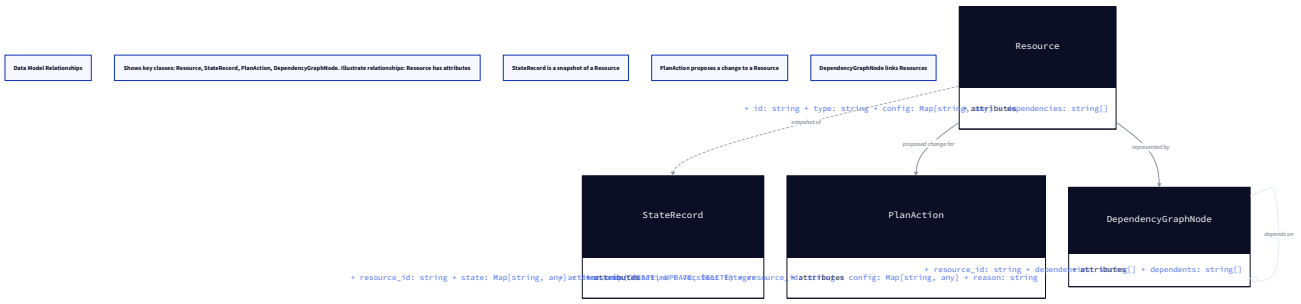
The data structures do not exist in isolation; they undergo a series of transformations as they move through the IaC engine's workflow. Understanding this lifecycle is key to understanding how the system maintains consistency and safety.

Lifecycle Stages and Transformations

The journey from configuration to deployed infrastructure involves three primary states for each resource, represented by our core types:

1. **Desired State (`Resource`):** Defined by the user in HCL/YAML.
2. **Proposed Change (`PlanAction`):** Calculated by comparing Desired State with Known State.
3. **Known State (`StateRecord`):** Persisted after successful application of a change.

The following diagram illustrates the relationships and transformations:



1. Configuration Parsing (Milestone 1) The **Parser** reads raw configuration files (HCL/YAML) and, after resolving variables, modules, and interpolations, produces a flat list of `Resource` objects. This list represents the complete desired state of the system. At this stage, `Resource.attributes` contain the final, user-specified values.

> **Example Transformation:** The parser encounters `instance_type = "${var.instance_size}"`. It resolves `var.instance_size` to `"t3.micro"` and stores `"t3.micro"` in the `Resource.attributes` map under the key `instance_type`.

2. State Loading and Diff Calculation (Milestone 2) The **State Manager** loads the previous `StateRecord` for each `Resource` from the state file (keyed by `resource_id`). For each resource, the system now holds two representations:

- **Desired (`Resource`):** `{id: "a", attributes: {"size": "large"}}`
- **Actual (`StateRecord`):** `{resource_id: "a", attributes: {"size": "medium"}}`

The **Planner** performs a diff between these two representations. The diff logic must be aware of provider-specific semantics (e.g., some attributes are immutable and force a `DELETE` + `CREATE` instead of an `UPDATE`). The output of this diff is a **draft** `PlanAction` for the resource.

> **Decision: Diffing Strategy**
>
> - **Context:** We need to determine what constitutes a change worthy of an `UPDATE` action. A naive comparison of the entire `attributes` map is insufficient because the state contains read-only/computed fields (like ARNs, IDs) that are not present in the configuration.
> - **Options Considered:**
>   1. **Full Deep Diff:** Compare all keys in both `Resource.attributes` and `StateRecord.attributes`, flagging any difference.
>   2. **Config-Key-Only Diff:** Only compare keys that are present in `Resource.attributes`. Ignore keys that only exist in `StateRecord.attributes` (computed fields).
>   3. **Schema-Aware Diff:** Use the provider's resource schema to classify attributes as "configurable" (requires update if changed) or "computed" (ignore for diff).
> - **Decision:** Option 2, **Config-Key-Only Diff**.
> - **Rationale:** It is simple to implement and correct for the majority of cases. Computed fields are outputs from the provider and should not trigger updates. While Option 3 is more robust, it requires integrating complex schema information at the planning stage, which is a significant increase in complexity for a learning project.
> - **Consequences:** This approach may incorrectly flag a change if a user removes a configurable attribute from their configuration (as the diff will see a mismatch between `config[attr]=None` and `state[attr]=value`). We must treat a `None` in config as "use default or remove," which requires special handling per attribute. This is a known simplification.

Option	Pros	Cons	Chosen?
Full Deep Diff	Simple, catches all changes.	Incorrectly triggers updates on computed/immutable fields, leading to failed applies.	❌
Config-Key-Only Diff	Avoids noise from computed fields. Simple logic.	Requires careful handling of `null`/absent values. May miss force-new updates on immutable fields.	✅
Schema-Aware Diff	Most accurate. Can handle immutable fields correctly.	Complex. Requires tight coupling with provider schemas early in the pipeline.	❌

3. Dependency Graph Construction and Plan Finalization (Milestone 3) The **Planner** builds a `DependencyGraph` from the list of resources. Each node is a `DependencyGraphNode`. Edges are created by analyzing `Resource` attribute values for references (e.g., `subnet_id = aws_subnet.main.id` creates a dependency from the referencing resource to `aws_subnet.main`) and processing explicit `depends_on` directives.

The draft `PlanAction` list is then **topologically sorted** based on this graph. `CREATE` actions are ordered from dependencies to dependents. `DELETE` actions are ordered in the **reverse** direction (dependents before dependencies). This sorted list becomes the final **execution plan**.

4. Plan Execution and State Persistence (Milestone 4) The **Executor** iterates through the sorted `PlanAction` list. For each action:

- It calls the appropriate CRUD method on the `BaseProvider` (e.g., `create(resource)` for `ActionType.CREATE`).
- Upon success, it calls the provider's `read` method to obtain the *actual* new attributes from the cloud API.
- It constructs a new `StateRecord` using these read attributes and the persisted dependency list.
- It updates the `PlanAction.new_state` field with this record.
- After all actions for a resource are complete, the **Executor** sends the updated `StateRecord` to the **State Manager**, which atomically writes it to the state file, replacing the old `StateRecord`.

This last step closes the loop: the `StateRecord` generated from execution becomes the new "known state" for the next run.

Walk-Through: A Complete Resource Lifecycle

Let's trace a single resource, `aws_instance.app_server`, through a typical create-update-delete scenario.

1. **Initial Run (Create):**
   - **Config:** `resource "aws_instance" "app_server" { ami = "ami-123" }`
   - **Parser Output:** `Resource{id="aws_instance.app_server", type="aws_instance", name="app_server", attributes={"ami": "ami-123"}}`
   - **State File:** No existing `StateRecord` for `aws_instance.app_server`.
   - **Planner Diff:** `prior_state` is `null`, `action_type` = `CREATE`.
   - **PlanAction:** `{action_type: CREATE, resource: Resource{...}, prior_state: null, new_state: StateRecord{predicted...}}`
   - **Executor:** Calls `provider.create(...)`. Gets actual instance ID `i-abc123` from cloud.
   - **New State:** `StateRecord{resource_id: "aws_instance.app_server", attributes: {"ami": "ami-123", "id": "i-abc123", "public_ip": "1.2.3.4"}, dependencies: []}` is written to state file.
2. **Second Run (Update):**
   - **Config Changed:** `ami = "ami-456"`
   - **Parser Output:** `Resource{attributes={"ami": "ami-456"}}`
   - **State Loaded:** The `StateRecord` from step 1.
   - **Planner Diff:** `ami` differs (`"ami-456"` vs `"ami-123"`). `action_type` = `UPDATE`.
   - **PlanAction:** `{action_type: UPDATE, resource: Resource{...ami-456}, prior_state: StateRecord{...ami-123}, new_state: StateRecord{predicted...}}`
   - **Executor:** Calls `provider.update(...)`. Cloud replaces the instance (new ID `i-def456`).
   - **New State:** `StateRecord{attributes: {"ami": "ami-456", "id": "i-def456", "public_ip": "5.6.7.8"}, ...}` overwrites the old record.
3. **Third Run (Delete):**
   - **Config:** The `aws_instance.app_server` block is removed.
   - **Parser Output:** No `Resource` with id `aws_instance.app_server`.
   - **State Loaded:** The `StateRecord` from step 2 still exists.
   - **Planner Diff:** Resource missing in config but present in state. `action_type` = `DELETE`.
   - **PlanAction:** `{action_type: DELETE, resource: Resource{last known config}, prior_state: StateRecord{...}, new_state: null}`
   - **Executor:** Calls `provider.delete(prior_state)`. Cloud destroys instance `i-def456`.

- **State Updated:** The `StateRecord` for `aws_instance.app_server` is removed from the state file.

This predictable lifecycle, governed by the clear transformation between `Resource`, `PlanAction`, and `StateRecord`, is what enables the IaC engine to be idempotent and safe.

Common Pitfalls

⚠️ Pitfall: Confusing `Resource.id` with Cloud Provider ID

- **The Mistake:** Storing the cloud-assigned identifier (e.g., `i-0abc123`) in the `Resource.id` field or using it as the key in the state file.
- **Why It's Wrong:** The `Resource.id` is a logical identifier from configuration. It must be immutable and known before any API call. The cloud ID is an *output* of the create operation and belongs in `StateRecord.attributes`. Using the cloud ID as a key would break state tracking if the resource is recreated (new cloud ID).
- **The Fix:** Always use the structured `{type}.{name}` format for `Resource.id` and `StateRecord.resource_id`. Store the cloud provider ID as a plain attribute (e.g., `attributes["id"]`) within the `StateRecord`.

⚠️ Pitfall: Storing Unresolved Variables or References in `Resource.attributes`

- **The Mistake:** The Parser puts raw interpolated strings like `"${var.region}"` or references like `"aws_vpc.main.id"` into the `Resource.attributes` map.
- **Why It's Wrong:** The Planner's diff logic and the Provider's CRUD methods cannot work with unresolved references. They need concrete values (like `"us-east-1"`) or opaque IDs (like `"vpc-123"`).
- **The Fix:** Ensure the **Parser** fully resolves all variable interpolations and module outputs *before* creating the final `Resource` objects. References to other resources should be resolved to their *state* values (e.g., the actual VPC ID) by the Planner when building the dependency graph, and these concrete values should be placed in a copy of the `Resource` used for planning.

⚠️ Pitfall: Forgetting to Update `dependencies` in `StateRecord`

- **The Mistake:** The `StateRecord` written after an apply contains an empty or outdated `dependencies` list.
- **Why It's Wrong:** The dependency graph is needed for `destroy` operations and for targeted plans. If the configuration is deleted, the system cannot rebuild the graph from source. An outdated list can lead to incorrect destroy order (trying to delete a subnet before deleting VMs inside it).
- **The Fix:** When the Planner builds the dependency graph, ensure the resulting `depends_on` list for each resource is captured and stored in the `StateRecord` that gets persisted after a successful apply.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Production)
Data Serialization	**Python's `json` module** (built-in, simple). Use `json.dumps` for writing and `json.loads` for reading.	**Pydantic models** with validation. Provides automatic type coercion, validation, and serialization.
In-Memory Data Structures	**Python dictionaries (`dict`) and lists.** Directly map to JSON structures. Use `dataclasses` or simple classes for type hints.	**TypedDict** (Python 3.8+) for stricter dictionary shape validation. `@dataclass` for mutable objects.
Enum Implementation	**Python `enum.Enum` class.** Provides clear names for `ActionType`.	**`str` -based enums** (`enum.Enum` with mixin) for easier JSON serialization.

B. Recommended File/Module Structure

Place the core data model definitions in a central module that all other components can import.

```
infrastructure-as-code-engine/
├── iac/
│   ├── __init__.py
│   ├── models/                 # Data model definitions
│   │   ├── __init__.py
│   │   ├── resource.py         # Resource class
│   │   ├── state.py            # StateRecord class & StateFile container
│   │   ├── plan.py             # PlanAction class & ActionType enum
│   │   └── graph.py            # DependencyGraphNode & Graph class
│   ├── parser/                 # Milestone 1
│   ├── state/                  # Milestone 2
│   ├── planner/                # Milestone 3
│   └── executor/               # Milestone 4
└── pyproject.toml
```

C. Infrastructure Starter Code (COMPLETE, ready to use)

1. Base Model Helper (JSON Serialization): Create a mixin class to easily convert our models to/from dictionaries for JSON serialization.

```python
# iac/models/__init__.py

import json

from typing import Any, Dict

from abc import ABC


class JsonSerializable(ABC):

    """Mixin class to provide to_dict and from_dict methods."""


    def to_dict(self) -> Dict[str, Any]:

        """Convert the object to a dictionary suitable for JSON serialization."""

        # This simple implementation works for dataclasses and simple classes.

        # For complex objects, override this method in the subclass.

        return self.__dict__.copy()


    @classmethod

    def from_dict(cls, data: Dict[str, Any]) -> Any:

        """Create an instance of the class from a dictionary."""

        return cls(**data)


    def to_json(self) -> str:

        """Serialize the object to a JSON string."""

        return json.dumps(self.to_dict(), indent=2)


    @classmethod

    def from_json(cls, json_str: str) -> Any:

        """Deserialize the object from a JSON string."""

        data = json.loads(json_str)

        return cls.from_dict(data)
```

2. Atomic File Operations Helper: Use this for safe reading/writing of state and lock files.

PYTHON

```python
# iac/utils/file.py

import os

import json

import tempfile

from pathlib import Path

from typing import Any


def write_atomic_json(filepath: Path, data: Any) -> None:

    """

    Write JSON data to a file atomically to prevent corruption.

    Writes to a temporary file in the same directory then renames it.

    """

    filepath = Path(filepath)

    # Create a temporary file in the same directory for atomic rename

    with tempfile.NamedTemporaryFile(

        mode='w',

        dir=filepath.parent,

        prefix=f".{filepath.name}.tmp.",

        delete=False

    ) as tmp:

        json.dump(data, tmp, indent=2)

        tmp.flush()

        os.fsync(tmp.fileno())

        tmp_path = Path(tmp.name)


    # Atomic rename (POSIX systems guarantee this is atomic)

    try:

        os.rename(tmp_path, filepath)

    except Exception:

        # Clean up temp file if rename fails

        tmp_path.unlink(missing_ok=True)

        raise


def read_json_with_backup(filepath: Path) -> Any:

    """

    Read JSON file, with automatic fallback to a backup if present.

    Returns the parsed data or raises FileNotFoundError if neither exists.

    """

    filepath = Path(filepath)

    backup_path = filepath.with_suffix(filepath.suffix + '.backup')
```

```
    data = None
    last_error = None

    # Try main file first
    if filepath.exists():
        try:
            with open(filepath, 'r') as f:
                data = json.load(f)
        except (json.JSONDecodeError, OSError) as e:
            last_error = e
            # Main file corrupted, try backup
            if backup_path.exists():
                try:
                    with open(backup_path, 'r') as f:
                        data = json.load(f)
                    # Backup was good, restore it
                    write_atomic_json(filepath, data)
                except (json.JSONDecodeError, OSError) as backup_error:
                    last_error = backup_error
                    data = None
    elif backup_path.exists():
        # No main file, but backup exists (e.g., after a crash)
        try:
            with open(backup_path, 'r') as f:
                data = json.load(f)
            # Restore from backup
            write_atomic_json(filepath, data)
        except (json.JSONDecodeError, OSError) as e:
            last_error = e
            data = None

    if data is None:
        raise FileNotFoundError(
            f"Could not read valid JSON from {filepath} or backup. Last error: {last_error}"
        )
    return data
```

D. Core Logic Skeleton Code

1. Data Model Definitions (`Resource` , `StateRecord` , `PlanAction` , `DependencyGraphNode`):

PYTHON

```python
# iac/models/resource.py

from dataclasses import dataclass, field

from typing import Any, Dict

from ..models import JsonSerializable


@dataclass

class Resource(JsonSerializable):

    """Represents a desired infrastructure resource from configuration."""

    id: str  # Format: "resource_type.resource_name"

    type: str  # e.g., "aws_instance"

    name: str  # Logical name from config

    attributes: Dict[str, Any] = field(default_factory=dict)


    def __post_init__(self):

        # Basic validation

        if not self.id:

            raise ValueError("Resource id cannot be empty")

        if '.' not in self.id:

            raise ValueError(f"Resource id should be in format 'type.name', got: {self.id}")
```

PYTHON

```python
# iac/models/state.py

from dataclasses import dataclass, field

from typing import Any, Dict, List, Optional

from ..models import JsonSerializable


@dataclass

class StateRecord(JsonSerializable):

    """Snapshot of a deployed resource's actual state."""

    resource_id: str

    resource_type: str

    resource_name: str

    attributes: Dict[str, Any] = field(default_factory=dict)

    dependencies: List[str] = field(default_factory=list)  # List of resource_ids this depends on


    def __post_init__(self):

        if not self.resource_id:

            raise ValueError("StateRecord.resource_id cannot be empty")
```

PYTHON

```python
# iac/models/plan.py

import enum

from dataclasses import dataclass, field

from typing import Optional

from .resource import Resource

from .state import StateRecord

from ..models import JsonSerializable


class ActionType(enum.Enum):

    """The type of change to perform on a resource."""

    CREATE = "create"

    UPDATE = "update"

    DELETE = "delete"

    NOOP = "noop"


    def __str__(self):

        return self.value


@dataclass

class PlanAction(JsonSerializable):

    """A proposed change to bring reality in line with desired state."""

    action_type: ActionType

    resource: Resource

    prior_state: Optional[StateRecord] = None

    new_state: Optional[StateRecord] = None


    def to_dict(self) -> Dict[str, Any]:

        # Custom serialization to handle Enums and nested objects

        data = {

            "action_type": self.action_type.value,

            "resource": self.resource.to_dict(),

            "prior_state": self.prior_state.to_dict() if self.prior_state else None,

            "new_state": self.new_state.to_dict() if self.new_state else None,

        }

        return data


    @classmethod

    def from_dict(cls, data: Dict[str, Any]) -> 'PlanAction':

        # Custom deserialization

        return cls(

            action_type=ActionType(data["action_type"]),
```

```
            resource=Resource.from_dict(data["resource"]),
            prior_state=StateRecord.from_dict(data["prior_state"]) if data.get("prior_state") else None,
            new_state=StateRecord.from_dict(data["new_state"]) if data.get("new_state") else None,
        )
```

PYTHON

```python
# iac/models/graph.py

from dataclasses import dataclass, field

from typing import Dict, List, Set

from ..models import JsonSerializable


@dataclass

class DependencyGraphNode(JsonSerializable):

    """A node in the resource dependency graph."""

    resource_id: str

    depends_on: List[str] = field(default_factory=list)  # Outgoing edges

    required_by: List[str] = field(default_factory=list)  # Incoming edges (derived)


class DependencyGraph:

    """A directed graph representing resource dependencies."""


    def __init__(self):

        self.nodes: Dict[str, DependencyGraphNode] = {}


    def add_node(self, resource_id: str) -> None:

        """Add a node for the given resource_id if it doesn't exist."""

        # TODO 1: Check if node with resource_id already exists in self.nodes

        # TODO 2: If not, create a new DependencyGraphNode with that resource_id

        # TODO 3: Store it in self.nodes

        pass


    def add_dependency(self, from_resource_id: str, to_resource_id: str) -> None:

        """

        Add a dependency edge: from_resource depends on to_resource.

        This means from_resource -> to_resource (from must be created after to).

        """

        # TODO 1: Ensure both nodes exist (call add_node for each)

        # TODO 2: Add to_resource_id to from_resource.depends_on list (if not already present)

        # TODO 3: Add from_resource_id to to_resource.required_by list (if not already present)

        pass


    def topological_sort(self) -> List[str]:

        """

        Return a list of resource_ids in topological order (dependencies first).

        Raises ValueError if a cycle is detected.

        """

        # TODO 1: Create a copy of the graph's dependency information

        # TODO 2: Find all nodes with zero in-degree (empty depends_on)
```

```
        # TODO 3: While there are zero in-degree nodes:
        #   TODO 3a: Remove one and add it to result list
        #   TODO 3b: For each node that required it, remove the dependency edge
        #   TODO 3c: Update zero in-degree list
        # TODO 4: If result list length != total nodes, there's a cycle → raise ValueError
        # TODO 5: Return the result list
        pass

    def reverse_topological_sort(self) -> List[str]:
        """
        Return a list of resource_ids in reverse topological order (dependents first).
        Useful for destroy operations.
        """
        # TODO 1: Get topological sort
        # TODO 2: Return it reversed
        pass
```

E. Language-Specific Hints

1. **Use `@dataclass`**: They automatically generate `__init__`, `__repr__`, and `__eq__` methods, making your data models clean and Pythonic.
2. **Type Hints**: Use them everywhere. They don't affect runtime but help with IDE autocompletion and catching bugs early with tools like `mypy`.
3. **JSON Serialization for Enums**: The built-in `json` module doesn't serialize `Enum` objects by default. Our `PlanAction.to_dict()` method manually converts the `ActionType` enum to its string value. Alternatively, you could write a custom JSON encoder.
4. **Default Factories**: Use `field(default_factory=list)` instead of `field(default=[])` for mutable defaults to avoid shared list instances across instances.

F. Milestone Checkpoint

After implementing the data models, verify they work correctly by creating a simple test script:

```python
# test_models.py

from iac.models import Resource, StateRecord, ActionType, PlanAction


# Create a Resource

r = Resource(id="aws_instance.web", type="aws_instance", name="web", attributes={"ami": "ami-123"})

print(f"Resource: {r}")

print(f"Resource JSON: {r.to_json()}")


# Create a StateRecord

s = StateRecord(

    resource_id="aws_instance.web",

    resource_type="aws_instance",

    resource_name="web",

    attributes={"id": "i-abc123", "ami": "ami-123"},

    dependencies=[]

)

print(f"\nStateRecord: {s}")


# Create a PlanAction

p = PlanAction(

    action_type=ActionType.UPDATE,

    resource=r,

    prior_state=s,

    new_state=None

)

print(f"\nPlanAction: {p}")

print(f"PlanAction JSON: {p.to_json()}")


# Test round-trip serialization

p2 = PlanAction.from_json(p.to_json())

assert p2.action_type == p.action_type

assert p2.resource.id == p.resource.id

print("\n✅ Round-trip serialization works!")
```

Run this script. Expected output should show the objects and their JSON representations without errors, confirming your basic data model is functional.

Component Design: Configuration Parser

> **Milestone(s):** 1

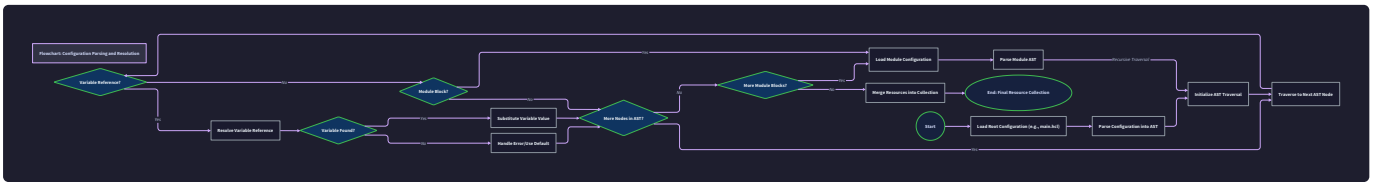
The **Configuration Parser** is the entry point of the IaC engine, responsible for translating human-readable configuration files into a normalized, machine-processable representation of desired infrastructure. It must comprehend a domain-specific language (like HCL or YAML), resolve dynamic expressions, assemble reusable modules, and produce a complete collection of `Resource` objects—the "desired state" that subsequent components will compare against reality. This component's accuracy is foundational; any misinterpretation here leads to incorrect plans and potentially destructive infrastructure changes.

Mental Model: The Interpreter and Expander

Imagine you are given a construction blueprint written in a mix of standard notations and custom shorthand. Your job is to produce a single, unambiguous, and fully detailed set of instructions for the construction crew. The original blueprint might have placeholders like "use the `main_color` specified in the `branding` folder," and it might reference entire pre-drawn sections from other blueprint files labeled "standard office module." You, as the interpreter, must:

1. **Parse the Syntax:** Understand the meaning of each symbol and notation (e.g., a box labeled "VM" means to provision a virtual machine).
2. **Expand Templates:** Find the definition of `main_color` (perhaps in a separate `variables.tf` file) and replace every placeholder with the concrete value "Deep Blue."
3. **Combine Chapters:** Locate the "standard office module" file, interpret its contents, and integrate its defined resources (desks, chairs, network jacks) into the main plan, possibly customizing them for this specific building.

In this analogy, the original HCL/YAML files are the blueprint with shorthand and references. The Parser acts as the **Interpreter and Expander**. It reads the raw text, understands the structure (parsing), fills in the blanks (variable interpolation), and pulls in external definitions (module resolution) to output one comprehensive, flat list of concrete `Resource` definitions—the final, unambiguous construction plan. This process is visualized in the parsing flowchart:



Interface and Public Methods

The Parser's public API is designed for a straightforward workflow: load a root configuration file and receive a complete set of resolved resources. Its core interface consists of the following methods.

Method Name	Parameters	Returns	Description
`parse_file`	`file_path: Path`	`dict` (Raw AST)	Reads a single configuration file from disk and parses it into a raw, unprocessed abstract syntax tree (AST) represented as a Python dictionary. This is a low-level method that does not perform resolution.
`resolve_variables`	`ast: dict, variable_files: List[Path], cli_vars: dict`	`dict` (Resolved AST)	Takes a raw AST and a set of variable values (from files and command-line overrides) and recursively replaces all variable interpolation expressions (e.g., `${var.name}`) with their concrete values. Returns a new AST with values resolved.
`load_module`	`module_call: dict, parent_dir: Path`	`dict` (Module AST)	Given a module block definition (from the AST) and the directory of the parent configuration, locates the module's source (local path), parses its files, recursively resolves its own variables and sub-modules, and returns the fully resolved AST for that module.
`process_configuration`	`root_file: Path, variable_files: List[Path], cli_vars: dict`	`List[Resource]`	**Primary entry point.** Orchestrates the full parsing pipeline. Parses the root file, resolves all variables, recursively loads all referenced modules, flattens the structure, and converts the final resolved AST into a list of `Resource` objects ready for the Planner.

The `Resource` type produced by `process_configuration` is a core data structure of the entire system. Its fields are detailed below.

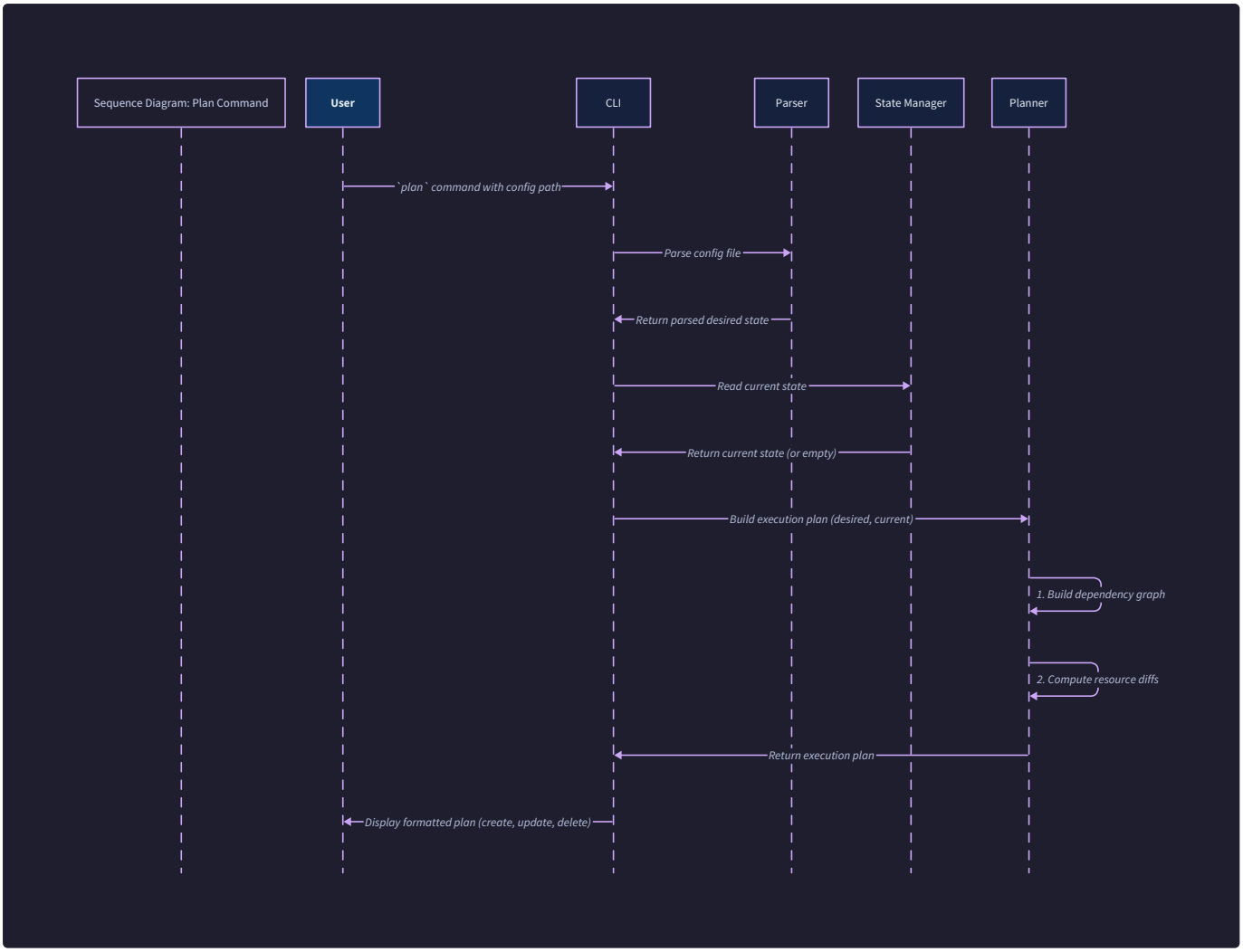
Field Name	Type	Description
`id`	`string`	A unique, deterministic identifier for the resource within the configuration. Typically formed as `<resource_type>.<resource_name>` (e.g., `aws_instance.web_server`).
`type`	`string`	The type of infrastructure resource, which maps to a specific provider and API (e.g., `aws_instance` , `google_sql_database_instance`).
`name`	`string`	The logical name given to this resource instance within the configuration, unique within its type scope.
`attributes`	`map[string]any`	A key-value map of all configuration arguments for the resource. These are the fully resolved, concrete values (e.g., `{"ami": "ami-12345", "instance_type": "t3.micro"}`).

Internal Behavior and Algorithm

The `process_configuration` method implements a multi-stage pipeline to transform text files into `Resource` objects. The following numbered algorithm details each step.

1. **Load and Parse Root Configuration:** Read the primary configuration file (e.g., `main.tf`) from disk. Pass its content through a syntax parser (for HCL or YAML) to produce a **raw AST**. This AST is a hierarchical dictionary mirroring the configuration's structure, containing unresolved blocks, expressions, and references.
2. **Extract and Merge Variable Values:** Gather variable definitions from all sources, following a precedence order (lowest to highest): default values in variable declarations, values from variable files (e.g., `terraform.tfvars`), and finally values provided via command-line flags (`-var`). Merge them into a single `variables` dictionary, with higher-priority sources overriding lower ones.
3. **Resolve Variable Interpolation:** Perform a depth-first traversal of the raw AST. For every string value encountered, check for the interpolation pattern `${...}`.
    1. If the pattern matches `var.<name>`, look up the concrete value in the merged `variables` dictionary and replace the entire expression with that value.
    2. If the pattern matches a more complex expression (e.g., `function.call`, `resource.attr`), tag it for later resolution by the Planner. For this milestone, we focus only on `var` references.
    3. Recursively process the updated value in case interpolation returns another string with nested expressions.
4. **Resolve Module Blocks:** Traverse the resolved AST to find all `module` blocks. For each module block:
    1. Extract the `source` attribute (a local file path).
    2. Compute the absolute source directory relative to the parent configuration's directory.
    3. Recursively call the parsing pipeline (`process_configuration`) on the module's own root file (e.g., `./modules/vpc/main.tf`), passing down any variables defined in the module block's `inputs`.
    4. The recursive call returns a list of `Resource` objects defined within that module.
    5. **Namespace the module's resources:** To avoid ID collisions, prepend the module's logical name to each returned resource's `id` and `name`. For example, a `vpc` module containing a resource `aws_vpc.main` becomes `module.vpc.aws_vpc.main`.
5. **Flatten and Convert to Resources:** After processing all modules, you have a collection of resolved resource blocks from the root and all nested modules. Convert each resource block into a `Resource` object.
    1. The `type` is the block label (e.g., `aws_instance`).
    2. The `name` is the user-defined label for that resource block.
    3. The `attributes` are the key-value pairs inside the block, now with all `var.*` interpolations resolved.
    4. The `id` is generated as a concatenation of `type` and `name` (and module prefix if applicable).
6. **Output Resource List:** Return the final, flat list of all `Resource` objects. This list represents the complete desired state of the infrastructure as declared in the configuration.

This sequence is part of the larger `plan` command workflow, as shown in the sequence diagram:



ADR: Abstract Syntax Tree vs. Direct Dict

Decision: Use a Lightweight Dict-based AST

- **Context:** We need an in-memory representation of the configuration's structure after parsing but before resolution. This representation must be easy to traverse, manipulate, and serialize. We must choose between building a formal Abstract Syntax Tree (AST) with dedicated node classes or using the native data structures (dicts, lists, strings) returned by the parser.
- **Options Considered:**
  1. **Formal AST Classes:** Define Python classes for each node type (e.g., `BlockNode`, `AttributeNode`, `VariableExprNode`). Each class has defined fields and methods for traversal and manipulation.
  2. **Lightweight Dict/List Structure:** Use the nested dictionaries and lists produced directly by the parser (e.g., PyHCL's output). Treat the structure generically.
- **Decision:** We will use the lightweight dict/list structure (Option 2).
- **Rationale:**
  - **Simplicity and Development Speed:** The dict structure requires no upfront class design. It maps directly to JSON, simplifying debugging and serialization for tests. For an educational project, this reduces cognitive load and lets us focus on the resolution algorithms.
  - **Parser Agnosticism:** Different parsers (for HCL, YAML, JSON) naturally output dicts. A generic dict interface allows us to swap or support multiple parsers without changing the resolution logic.
  - **Sufficient for Needs:** Our resolution algorithms (tree traversal, string replacement) are structurally simple. They don't require the type safety or encapsulation benefits of a full class hierarchy at this stage.
- **Consequences:**
  - **Pros:** Faster to implement, easier to debug (`print(json.dumps(ast, indent=2))`), flexible for different input formats.
  - **Cons:** Loss of type safety—the code must defensively check for expected keys and structures. Less self-documenting than a class-based model. Complex transformations in future extensions might become harder to manage.

The comparison table below summarizes the trade-offs.

Option	Pros	Cons	Chosen?
Formal AST Classes	Type-safe, self-documenting, enables visitor pattern for complex operations, compile-time checks possible.	Higher upfront design cost, more boilerplate code, tightly coupled to a specific parser's output structure.	No
Lightweight Dict/List	Rapid prototyping, easy serialization/debugging, parser-agnostic, less code to maintain.	No type checking, reliance on string keys, potential for runtime errors due to malformed structure.	**Yes**

Common Pitfalls

⚠️ Pitfall: Infinite Loop in Variable Resolution

- **Description:** A variable's value is defined by referencing another variable, which in turn references the first, either directly (`var.a -> var.b -> var.a`) or through a longer chain. During the `resolve_variables` traversal, the parser gets stuck in an endless loop.
- **Why it's Wrong:** The algorithm will never terminate, consuming CPU and hanging the IaC engine. This violates the requirement for predictable execution.
- **How to Fix:** Implement a **dependency graph for variables** before resolution. Before replacing values, analyze all variable expressions to build a graph of `var.a` depends on `var.b` . Run a cycle detection algorithm (like Depth-First Search with coloring). If a cycle is found, abort with a clear error message pointing to the involved variables.

⚠️ Pitfall: The `count` / `for_each` Interpolation Chicken-Egg Problem

- **Description:** The `count` meta-argument determines how many instances of a resource are created. If `count` itself uses a variable interpolation (e.g., `count = var.num_servers`), but that variable's value is derived from the output attribute of another resource (e.g., `num_servers = length(aws_autoscaling_group.web.instances)`), you have a paradoxical dependency. You need the resource's output to know how many of it to create.
- **Why it's Wrong:** This creates a circular dependency that cannot be resolved at parse time. The parser cannot resolve `var.num_servers` because its value is unknown until *after* some infrastructure is provisioned.
- **How to Fix:** Recognize that `count` and `for_each` are **runtime evaluations**. The parser should leave interpolations within these meta-arguments unresolved during the initial parsing phase. They become **dynamic references** that are evaluated by the Planner *after* the dependency graph is built, using values from the state file or from outputs of already-applied resources during the `apply` phase.

⚠️ Pitfall: Module Source Path Ambiguity

- **Description:** A module's `source` attribute is specified as a relative path like `../modules/vpc` . If the parser does not properly resolve this path relative to the calling configuration's directory, it may load the wrong files or fail to find them.
- **Why it's Wrong:** Leads to `FileNotFoundError` or, worse, silently loading an unrelated configuration, resulting in a plan that doesn't match the user's intent.
- **How to Fix:** Always resolve module source paths to absolute paths as early as possible. Use the parent configuration file's directory (`parent_dir`) as the base. The rule is: `abs_source = (parent_dir / source).resolve()` . Document this resolution strategy clearly.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option (Recommended for Learning)	Advanced Option (For Production)
HCL Parser	`pyhcl2` (a Python port of HCL 2) - Pure Python, easier to install and debug.	`python-hcl2` - Another popular parser. For maximum fidelity, use the official **HashiCorp Go libraries** via a subprocess or CFFI, but this adds complexity.
YAML Parser	Python's built-in `yaml` module (`PyYAML` if not installed).	Use a schema validator like `yamale` alongside parsing to enforce structure.
Data Structure	Nested `dict` and `list` objects.	Use `dataclasses` or `pydantic` models for the final `Resource` list, while keeping the intermediate AST as dicts.

B. Recommended File/Module Structure Place the parser logic in a dedicated module. Separate concerns for file I/O, syntax parsing, and resolution logic.

```
iac-engine/
├── iac_engine/
│   ├── __init__.py
│   ├── cli.py                    # CLI commands (plan_command, apply_command)
│   ├── parser/                   # Configuration Parser Component
│   │   ├── __init__.py
│   │   ├── core.py               # Parser class with main interface methods
│   │   ├── resolvers.py          # Variable and module resolution logic
│   │   ├── file_loader.py        # Safe file loading utilities (starter code below)
│   │   └── exceptions.py         # Parser-specific exceptions (e.g., ParseError, CyclicVariableError)
│   ├── models.py                 # Shared data models (Resource, StateRecord, etc.)
│   └── ... (other components)
├── configs/                      # Example configurations for testing
│   ├── main.tf
│   └── variables.tfvars
├── pyproject.toml
└── README.md
```

C. Infrastructure Starter Code

The following is a complete, reusable utility for safely reading configuration files with basic error handling. It should be placed in `parser/file_loader.py`.

PYTHON

```python
"""
Safe file loading utilities for the configuration parser.
"""
import json
from pathlib import Path
from typing import Any, Union
import yaml
import hcl2  # Requires: pip install pyhcl2

class FileLoaderError(Exception):
    """Base exception for file loading failures."""
    pass

def safe_read_file(filepath: Path) -> str:
    """
    Read a text file safely with explicit error handling.

    Args:
        filepath: Path to the file.

    Returns:
        The file content as a string.

    Raises:
        FileLoaderError: If the file cannot be read (does not exist, permission denied, etc.).
    """
    try:
        return filepath.read_text(encoding='utf-8')
    except (FileNotFoundError, PermissionError, IsADirectoryError, UnicodeDecodeError) as e:
        raise FileLoaderError(f"Failed to read file '{filepath}': {e}") from e

def load_config_file(filepath: Path) -> dict:
    """
    Load and parse a configuration file based on its extension.
    Supports .tf (HCL), .tf.json (JSON), .yaml, .yml.

    Args:
        filepath: Path to the configuration file.

    Returns:
        A dictionary representing the parsed configuration (raw AST).

    Raises:
```

```
        FileLoaderError: If the file cannot be parsed or has an unsupported extension.
    """
    content = safe_read_file(filepath)
    suffix = filepath.suffix.lower()

    try:
        if suffix == '.tf':
            # Parse HCL using pyhcl2
            return hcl2.loads(content)
        elif suffix == '.json' or filepath.name.endswith('.tf.json'):
            return json.loads(content)
        elif suffix in ('.yaml', '.yml'):
            # Using SafeLoader by default to avoid arbitrary code execution
            return yaml.safe_load(content)
        else:
            raise FileLoaderError(f"Unsupported configuration file extension: {suffix}")
    except (hcl2.HCL2Error, json.JSONDecodeError, yaml.YAMLError) as e:
        raise FileLoaderError(f"Failed to parse '{filepath}': {e}") from e
```

D. Core Logic Skeleton Code

The main parsing logic resides in `parser/core.py`. Below is the skeleton for the primary `Parser` class with detailed TODO comments that map directly to the algorithm steps described earlier.

PYTHON

```python
"""
Core configuration parser implementing the parsing pipeline.
"""
from pathlib import Path
from typing import Dict, List, Any, Optional
from ..models import Resource
from . import resolvers
from .file_loader import load_config_file, FileLoaderError

class Parser:
    """Orchestrates the parsing of IaC configuration files."""

    def parse_file(self, file_path: Path) -> dict:
        """
        Parse a single configuration file into a raw AST (dict).

        Args:
            file_path: Path to the .tf, .yaml, or .json file.

        Returns:
            A dictionary representing the raw, unprocessed AST.
        """
        # TODO 1: Use the `load_config_file` utility to load and parse the file.
        # TODO 2: Return the parsed dictionary.
        pass

    def resolve_variables(self, ast: dict, variable_files: List[Path], cli_vars: Dict[str, Any]) -> dict:
        """
        Resolve variable interpolations within an AST.

        Args:
            ast: The raw AST dictionary.
            variable_files: List of paths to variable definition files (.tfvars, .json, .yaml).
            cli_vars: Dictionary of variable values provided via command line.

        Returns:
            A new AST dictionary with variable references replaced by concrete values.
        """
        # TODO 1: Extract default variable declarations from the 'variable' blocks in the AST.
        # TODO 2: Load variable values from each file in `variable_files` (using load_config_file).
        # TODO 3: Merge values: defaults < file values < cli_vars (highest priority).
        # TODO 4: Perform a depth-first traversal of the AST.
```

```
        # TODO 5: For each string value found, check for the pattern ${var.<name>}.
        # TODO 6: Look up the variable name in the merged values dictionary.
        # TODO 7: Replace the entire interpolation expression with the concrete value.
        # TODO 8: (Advanced) Detect circular dependencies between variables and raise an error.
        # Hint: Use a recursive function or a stack for traversal.
        pass

    def load_module(self, module_call: dict, parent_dir: Path) -> List[Resource]:
        """
        Load and fully resolve a module, returning its resources.

        Args:
            module_call: The AST of the 'module' block (contains 'source', 'inputs', etc.).
            parent_dir: The directory of the configuration file containing this module call.

        Returns:
            A list of Resource objects defined within the module, with names prefixed.
        """
        # TODO 1: Extract the 'source' attribute from module_call. Raise error if missing.
        # TODO 2: Resolve the source path to an absolute directory relative to parent_dir.
        # TODO 3: Find the main configuration file within the module directory (e.g., main.tf).
        # TODO 4: Call `self.process_configuration` on that main file, passing the 'inputs' as cli_vars.
        # TODO 5: For each Resource returned, prepend the module's logical name to its `id` and `name`.
        #   Example: module name = "vpc", resource id = "aws_vpc.main" -> "module.vpc.aws_vpc.main"
        # TODO 6: Return the list of prefixed resources.
        pass

    def process_configuration(self, root_file: Path,
                              variable_files: Optional[List[Path]] = None,
                              cli_vars: Optional[Dict[str, Any]] = None) -> List[Resource]:
        """
        Main entry point: parse a root configuration and all its modules.

        Args:
            root_file: Path to the root configuration file.
            variable_files: Optional list of variable definition files.
            cli_vars: Optional dictionary of command-line variable overrides.

        Returns:
            A flat list of all Resource objects from the root and nested modules.
        """
        # TODO 1: Initialize variable_files and cli_vars to empty lists/dicts if None.
```

```
    # TODO 2: Parse the root file using `self.parse_file`.
    # TODO 3: Resolve variables in the root AST using `self.resolve_variables`.
    # TODO 4: Initialize an empty list `all_resources`.
    # TODO 5: Traverse the resolved AST to find 'resource' blocks at the root level.
    # TODO 6: For each root resource block, convert it to a Resource and add to all_resources.
    # TODO 7: Traverse the resolved AST to find 'module' blocks.
    # TODO 8: For each module block, call `self.load_module` and extend all_resources with the result.
    # TODO 9: Return the complete list of resources.
    pass
```

E. Language-Specific Hints

- **Recursive Traversal:** When implementing `resolve_variables`, a recursive helper function that handles dictionaries, lists, and strings is simpler than managing an explicit stack for this use case.
- **Path Handling:** Always use `pathlib.Path` objects instead of string concatenation for file paths. Use `/` operator to join paths (e.g., `parent_dir / "modules/vpc"`).
- **Module Discovery:** A simple heuristic for finding a module's main file is to look for `main.tf`, then `variables.tf`, then any `.tf` file in the module directory.
- **Error Messages:** When raising exceptions, include the file name, line number (if available from the parser), and the specific problematic construct (e.g., "Undefined variable 'region' in main.tf").

F. Milestone Checkpoint

After implementing the `Parser` class, you should be able to run the following verification steps:

1. **Unit Test:** Create a simple test configuration.

```python
# test_parser.py (simplified example)
from iac_engine.parser.core import Parser
from pathlib import Path

parser = Parser()
resources = parser.process_configuration(
    root_file=Path("configs/main.tf"),
    variable_files=[Path("configs/terraform.tfvars")],
    cli_vars={"instance_count": 2}
)
print(f"Parsed {len(resources)} resources")
for r in resources:
    print(f"  - {r.id}: {r.attributes}")
```

2. **Expected Behavior:** Given a valid configuration, the script should print a list of `Resource` objects with fully resolved attribute values (no `${var...}` placeholders). Module resources should have IDs prefixed with `module.<name>.`.
3. **Signs of Trouble:**
    - `FileLoaderError`: Check file paths and ensure the parser's working directory is correct.
    - **Variable not resolved:** Verify the variable merging logic and the traversal/replacement in `resolve_variables`.
    - **Module resources missing:** Ensure `load_module` is being called and that it correctly recurses into subdirectories.

Component Design: State Manager

> **Milestone(s):** 2

The **State Manager** is the system of record for the IaC engine, maintaining a persistent, authoritative snapshot of what infrastructure currently exists according to the system's knowledge. It bridges the declarative world of configuration files and the imperative reality of cloud resources by storing the last known state after each successful operation. Beyond simple persistence, it ensures safe concurrent access through locking mechanisms and provides the critical diffing logic that determines what changes are needed.

Mental Model: The Ledger and Lock

Think of the State Manager as a combination of a **bank ledger** and a **safety deposit box lock**. The ledger (the state file) is a meticulously kept record of every resource the system has deployed: its unique cloud identifier, its current attributes, and how resources are connected. This ledger allows the system to answer the fundamental question: "What do I already have?"

The safety deposit box lock (the state lock) protects this ledger during updates. Imagine multiple people (or automated processes) trying to update the same bank ledger simultaneously—chaos would ensue. The lock ensures only one "teller" can modify the ledger at a time, preventing conflicting updates that could corrupt the record or cause duplicate resource creation. When a teller starts an update, they take the key (acquire the lock), make their changes, and then return the key (release the lock) for the next person.

This dual role—persistent record-keeping and concurrent access control—makes the State Manager a critical linchpin for safety and correctness in any multi-user or automated IaC environment.

Interface and Public Methods

The State Manager exposes a clean interface centered around four core operations: reading/writing state, acquiring/releasing locks, and computing the difference between states. The following table details the public contract.

Method Name	Parameters	Returns	Description
`read_state`	`state_path: Path`	`Dict[str, StateRecord]`	Loads the state file from the given path, parsing its JSON content into an in-memory dictionary keyed by resource address (e.g., `"aws_instance.web"`). Handles missing files (returns empty dict) and corrupted content (attempts recovery from backup).
`write_state`	`state_path: Path`, `state_data: Dict[str, StateRecord]`	`None`	Persists the in-memory state dictionary to disk at the specified path. Performs an **atomic write** to prevent corruption: writes to a temporary file, syncs to disk, then renames to the target path. Optionally creates a backup of the previous state file.
`acquire_lock`	`lock_path: Path`, `timeout_seconds: int = 30`, `heartbeat_interval: int = 10`	`LockHandle`	Attempts to acquire an exclusive lock for the state. Creates a lock file containing process metadata (PID, hostname, timestamp). If the lock already exists, checks if it's stale (older than timeout). Returns a handle object that must be used to release the lock. Starts a background heartbeat thread to periodically update the lock timestamp, preventing it from becoming stale during long operations.
`release_lock`	`lock_handle: LockHandle`	`None`	Releases the lock represented by the given handle. Stops the heartbeat thread and deletes the lock file. Must be called even if errors occur during apply to prevent deadlocks.
`compute_diff`	`current_state: Dict[str, StateRecord]`, `desired_resources: List[Resource]`	`Dict[str, PlanAction]`	The core diffing engine. Compares the current deployed state (from `read_state`) with the desired state (from the Parser). For each desired resource, determines if it needs to be created, updated, deleted, or left alone (`NOOP`). Returns a dictionary of actions keyed by resource address.
`get_resource_address`	`resource: Resource`	`str`	Helper method that generates a unique address string for a resource, following the convention `"{resource_type}.{resource_name}"`. This address is used as the key in the state dictionary and for dependency tracking.

Supporting Data Types:

Type Name	Fields	Description
`StateRecord`	`resource_id: str`, `resource_type: str`, `resource_name: str`, `attributes: Dict[str, Any]`, `dependencies: List[str]`	A snapshot of a resource as it exists in the cloud. The `resource_id` is the unique identifier assigned by the cloud provider (e.g., AWS instance ID `i-12345`). `attributes` are the *actual* values read from the cloud API at the last refresh. `dependencies` is a list of resource addresses this resource depends on, used for graph construction.
`LockHandle`	`lock_path: Path`, `process_id: int`, `lock_id: str` (UUID), `heartbeat_thread: Optional[Thread]`, `stop_event: Event`	An opaque object representing an acquired lock. It holds references to the lock file path, the current process's ID, a unique lock identifier for verification, and the background heartbeat thread with its stop signal. The caller should not modify this object.
`PlanAction`	`action_type: ActionType`, `resource: Resource`, `prior_state: Optional[StateRecord]`, `new_state: Optional[StateRecord]`	The output of `compute_diff`. Describes a change to be made. For `CREATE`, `prior_state` is `None`. For `DELETE`, `new_state` is `None`. For `UPDATE`, both are populated with the old and new states.

Internal Behavior and Algorithm

The State Manager's internal logic can be decomposed into three primary algorithms: atomic state persistence, lock management with heartbeats, and state diff calculation.

1. Atomic State Persistence with `write_atomic_json`

To guarantee the state file is never left in a partially written, corrupted state, the write operation follows an **atomic rename pattern**. This is implemented in the helper function `write_atomic_json`, which the `write_state` method calls.

1. **Serialize to JSON**: Convert the state dictionary (containing `StateRecord` objects) into a JSON string. Use a consistent indentation for readability.
2. **Create Temporary File**: Generate a temporary file path in the same directory as the target state file, using a prefix (e.g., `.terraform_state.tmp-`) and a random suffix to avoid collisions.
3. **Write and Sync**: Open the temporary file in write mode (`'w'`). Write the JSON string. Call `os.fsync()` on the file descriptor to ensure the data is flushed to physical disk, not just the OS cache.
4. **Create Backup (Optional)**: If the target state file already exists, rename it to a backup file (e.g., `.terraform_state.backup`) before the next step. This provides a recovery point.
5. **Atomic Rename**: Perform an atomic filesystem rename operation (`os.rename`) to move the temporary file to the final state file path. On POSIX systems, rename is atomic even in the face of crashes: the target path will point either to the old file or the new file, never a mixture.
6. **Cleanup**: If the backup file exists and the rename succeeded, the old backup (if any) can be removed. If any step fails, the temporary file is cleaned up, leaving the original state file intact.

2. Lock Acquisition with Heartbeat

The locking mechanism prevents concurrent modifications. It uses a simple file-based lock with a heartbeat to handle long-running operations and process crashes.

1. **Check for Existing Lock**: When `acquire_lock` is called, it first checks if a lock file exists at `lock_path`.
2. **Stale Lock Detection**: If a lock file exists, read its JSON content (containing `process_id`, `hostname`, `timestamp`, and `lock_id`). Calculate the age of the lock (current time minus `timestamp`). If the age exceeds the `timeout_seconds` parameter, the lock is considered **stale** (likely from a crashed process). Log a warning and proceed to steal the lock. Otherwise, wait and retry (with a backoff) until the lock is released or becomes stale.
3. **Create New Lock**: If no lock exists or the existing lock is stale, create a new lock file. Write a JSON object with the current process's PID, the machine's hostname, the current timestamp, and a newly generated UUID (`lock_id`). Perform an atomic write (similar to state writes) to ensure the lock file is created completely.
4. **Start Heartbeat Thread**: Create a background daemon thread that will periodically (every `heartbeat_interval` seconds) update the lock file's timestamp to the current time. This "refreshes" the lock, preventing it from appearing stale during a long `apply` operation. The thread runs until a stop event is set.
5. **Return Handle**: Package the lock file path, PID, `lock_id`, and references to the heartbeat thread and stop event into a `LockHandle` object and return it.

3. Computing the State Diff

The `compute_diff` algorithm is the brain of the planning phase. It determines the set of actions required to converge the current state to the desired state.

1. **Index Current State**: Build a dictionary mapping resource addresses (`type.name`) to their `StateRecord` objects from the `current_state` input.
2. **Index Desired Resources**: Build a similar dictionary for the `desired_resources` list, using the `get_resource_address` helper.
3. **Initialize Action Dictionary**: Create an empty dictionary to hold `PlanAction` objects, keyed by resource address.

4. **Iterate Over Desired Resources**: For each resource in the desired resources dictionary:
    - **Address Lookup**: Look up the resource's address in the current state index.
    - **No Current State (`CREATE`)**: If not found, the resource does not exist. Create a `PlanAction` with `action_type=ActionType.CREATE`, the `resource` set to the desired resource, `prior_state=None`, and `new_state` set to a `StateRecord` populated from the desired resource (but with `resource_id` initially empty).
    - **Current State Exists**: If found, compare the desired resource's attributes with the current state's attributes. Perform a **deep, semantic comparison**. For complex nested structures (like security group rules), order may not matter. If any attribute differs (considering provider-specific semantics), mark the resource for `UPDATE`. Create a `PlanAction` with the changed resource and both states. If no attributes differ, mark as `NOOP`.
5. **Identify Resources to Delete**: Find all resources in the current state index that are *not* present in the desired resources index. For each, create a `PlanAction` with `action_type=ActionType.DELETE`, `resource` constructed from the `StateRecord` (type and name), `prior_state` set to the record, and `new_state=None`.
6. **Return Actions**: Return the populated action dictionary.

> **Key Design Insight:** The diff algorithm must be **idempotent**. Running `plan` twice on the same configuration and state should produce an identical plan (ideally empty after an apply). This requires the comparison logic to ignore transient or read-only fields (like `arn` in AWS, which is derived from the ID) and to handle default values that the provider may inject.

ADR: Pessimistic File Locking vs. Optimistic Concurrency

> **Decision: Use Pessimistic File Locking for State Modification**
>
> - **Context**: The state file is a shared resource that must be modified by concurrent `apply` commands (from multiple users or CI/CD jobs). Without coordination, simultaneous writes could corrupt the file or cause race conditions where one process overwrites another's changes, leading to resource duplication or orphaned resources.
> - **Options Considered**:
>   1. **Pessimistic File Locking (Chosen)**: Acquire an exclusive lock (via a lock file) before reading the state, hold it throughout the `plan` and `apply` workflow, and release only after the new state is written. This ensures serializability.
>   2. **Optimistic Concurrency Control (OCC)**: Read the state without a lock, perform the plan and apply, then attempt to write the new state only if the state file's content hasn't changed since the initial read (using a version stamp or checksum). If it has changed, abort and retry from the beginning.
>   3. **Distributed Lock Service**: Use an external service like Redis, etcd, or a database to manage distributed locks, which is more robust in highly concurrent, multi-host environments.
> - **Decision**: Implement a simple **pessimistic file lock** using a lock file with a heartbeat mechanism and stale lock detection.
> - **Rationale**:
>   - **Simplicity and Educational Value**: File locks are easier to understand and implement correctly for a learning project. The concepts of atomic file operations, lock files, and heartbeats are fundamental.
>   - **Deterministic Behavior**: Pessimistic locking provides a clear, linear workflow: acquire lock → plan → apply → write → release. There's no need for complex retry loops or merge conflict resolution logic.
>   - **Adequate for Common Use Case**: For many small to medium teams, serializing `apply` operations is acceptable and even desirable to prevent unexpected interactions.
> - **Consequences**:
>   - **Positive**: Implementation is relatively straightforward. Provides strong consistency guarantees. Lock file and heartbeat logic are reusable patterns.
>   - **Negative**: Can cause bottlenecks if many `apply` operations are queued. Requires careful handling of process crashes to avoid stale locks (addressed via timeout+heartbeat). Not suitable for distributed workflows across multiple machines without a shared filesystem.

Option	Pros	Cons	Chosen?
Pessimistic File Locking	Simple to implement; strong consistency; clear workflow; good for learning	Can bottleneck throughput; requires shared filesystem for team; stale lock handling needed	**Yes**
Optimistic Concurrency Control	Allows parallel planning; no waiting for locks; better for high concurrency	Complex retry/merge logic; "plan" may be invalidated, wasting work; harder to debug	No
Distributed Lock Service	Robust for distributed teams; no shared filesystem requirement; advanced features (leases)	Introduces external dependency; operational overhead; more complex to implement	No

Common Pitfalls

⚠️ Pitfall: Partial Write Corrupts State File

- **Description**: If the system crashes or is killed while writing the state file (e.g., during a power loss), the file may be left with partially written JSON, making it unreadable.
- **Why It's Wrong**: A corrupted state file breaks the entire system. The engine can no longer determine what resources it manages, leading to potential resource orphanage or attempted duplicate creation.
- **How to Fix**: Always use the **atomic rename pattern** (`write_atomic_json`). Write the complete new state to a temporary file, `fsync` it, then rename it over the old state file. The rename is an atomic operation at the filesystem level. Additionally, keep a backup of the previous state file to allow manual recovery.

⚠️ Pitfall: Stale Lock Blocks All Operations

- **Description**: A process acquires a lock and then crashes (or is killed) before releasing it. The lock file remains on disk, preventing any other process from acquiring the lock, effectively halting all deployments.
- **Why It's Wrong**: This creates a single point of failure. Manual intervention (deleting the lock file) is required to resume operations, which is error-prone and may lead to concurrent applies if done incorrectly.
- **How to Fix**: Implement **stale lock detection**. Store a timestamp in the lock file and have processes check the lock's age. If a lock is older than a configurable timeout (e.g., 30 minutes), a new process can assume the owner is dead, log a warning, and "break" the stale lock. Complement this with a **heartbeat** for long-running operations, where the lock owner periodically updates the timestamp to signal it's still alive.

⚠️ Pitfall: Remote State Race Condition (Read-Modify-Write)

- **Description**: When using a remote backend (like S3), two processes might concurrently read the same state file (version 1). Both compute plans based on version 1. The first process applies changes and writes back state version 2. The second process, unaware of the update, applies its (now possibly invalid) changes and overwrites with a state derived from version 1, obliterating the first process's changes.
- **Why It's Wrong**: This is a classic lost update problem. It can cause resource conflicts, configuration rollback, or hidden dependencies being broken.
- **How to Fix**: For remote backends, you **must** implement optimistic concurrency control or use the remote system's locking capabilities. Use a version identifier (ETag in S3, conditional writes) in the state metadata. On write, include a condition that the remote object's version must match what was read. If the condition fails, abort and require the user to retry with the updated state. *Note: Our initial implementation uses local files, but this pitfall is critical for the "Remote State Backend" deliverable.*

Implementation Guidance

A. Technology Recommendations

Component	Simple Option (Recommended)	Advanced Option (Future)
State Serialization	JSON via Python's `json` module (human-readable, debuggable)	Protocol Buffers or CBOR (smaller, faster, binary)
Atomic File Writes	`tempfile.NamedTemporaryFile` + `os.replace` (Python 3.3+)	Platform-specific syscalls (`fsync` , `rename`)
Locking	File-based lock with `fcntl` (Unix) / `msvcrt.locking` (Windows)	Distributed lock via Redis/etcd API
Remote Backend	AWS S3 with Boto3, using versioned buckets	Custom HTTP backend with auth, caching

B. Recommended File/Module Structure

```
iac_engine/
├── iac_engine/
│   ├── __init__.py
│   ├── cli.py                    # plan_command, apply_command
│   ├── parser/                   # Milestone 1
│   ├── state/                    # State Manager (this component)
│   │   ├── __init__.py
│   │   ├── manager.py            # StateManager class (main logic)
│   │   ├── locking.py            # LockHandle, file locking utilities
│   │   ├── backends/             # For remote state (local, s3, http)
│   │   │   ├── __init__.py
│   │   │   ├── local.py
│   │   │   └── s3.py
│   │   └── serialization.py      # StateRecord helper methods (to/from dict)
│   ├── planner/                  # Milestone 3
│   ├── executor/                 # Milestone 4
│   └── providers/                # Milestone 4
└── tests/
    └── state/
        ├── test_manager.py
        └── test_locking.py
```

C. Infrastructure Starter Code

Below is complete, reusable code for atomic file operations and a simple lockfile implementation. Learners should place these in `state/serialization.py` and `state/locking.py` respectively.

PYTHON

```python
# iac_engine/state/serialization.py

"""Utilities for atomic file operations and JSON serialization."""

import json

import os

import tempfile

from pathlib import Path

from typing import Any


def write_atomic_json(filepath: Path, data: Any) -> None:

    """

    Write JSON data to a file atomically to prevent corruption.



    Args:

        filepath: The target file path.

        data: Any JSON-serializable Python object.



    Raises:

        OSError: If file operations fail.

        TypeError: If data is not JSON serializable.

    """

    # Create a temporary file in the same directory for atomic rename

    temp_fd, temp_path = tempfile.mkstemp(

        prefix=f"{filepath.name}.tmp.",

        suffix=".json",

        dir=filepath.parent,

        text=True

    )

    try:

        with os.fdopen(temp_fd, 'w') as f:

            json.dump(data, f, indent=2, default=str)

            f.flush()

            os.fsync(f.fileno())  # Force write to disk

        # Atomic rename (replace if exists, works cross-platform)

        os.replace(temp_path, filepath)

    except Exception:

        # Clean up temp file on any error

        try:

            os.unlink(temp_path)

        except OSError:
```

```
            pass
        raise

def read_json_with_backup(filepath: Path) -> Any:
    """
    Read JSON file, with automatic fallback to a backup if present.

    Args:
        filepath: The primary state file path.

    Returns:
        The parsed JSON data (typically a dict). Returns an empty dict
        if neither the primary nor backup file exists.

    Raises:
        json.JSONDecodeError: If both primary and backup files contain
            invalid JSON (after attempting to read backup).
    """
    backup_path = filepath.parent / f"{filepath.name}.backup"

    for path in [filepath, backup_path]:
        if path.exists():
            try:
                with open(path, 'r') as f:
                    return json.load(f)
            except json.JSONDecodeError:
                # Log warning but try backup if available
                print(f"Warning: Corrupted JSON in {path}, trying backup...")
                continue
    # Neither file exists or both are corrupt
    return {}
```

PYTHON

```python
# iac_engine/state/locking.py

"""File-based locking with stale lock detection and heartbeat."""

import json

import os

import threading

import time

import uuid

from datetime import datetime

from pathlib import Path

from typing import Optional


class LockHandle:

    """Represents an acquired lock."""

    def __init__(self, lock_path: Path, lock_id: str):

        self.lock_path = lock_path

        self.lock_id = lock_id  # UUID to verify we own the lock

        self.heartbeat_thread: Optional[threading.Thread] = None

        self.stop_event = threading.Event()


class FileLock:

    """Manages acquisition and release of a file-based lock."""

    def __init__(self, lock_path: Path, timeout_seconds: int = 300):

        self.lock_path = lock_path

        self.timeout_seconds = timeout_seconds



    def acquire(self, heartbeat_interval: int = 30) -> LockHandle:

        """

        Acquire an exclusive lock, with stale lock detection.



        Args:

            heartbeat_interval: How often (seconds) to refresh the lock timestamp.



        Returns:

            A LockHandle that must be used to release the lock.



        Raises:

            TimeoutError: If unable to acquire lock within timeout.

        """

        start_time = time.time()

        lock_id = str(uuid.uuid4())
```

```
    while time.time() - start_time < self.timeout_seconds:
        if not self.lock_path.exists():
            # No lock file, try to create it
            if self._create_lock_file(lock_id):
                handle = LockHandle(self.lock_path, lock_id)
                self._start_heartbeat(handle, heartbeat_interval)
                return handle

        # Lock exists, check if stale
        lock_data = self._read_lock_file()
        if lock_data:
            lock_age = time.time() - lock_data['timestamp']
            if lock_age > self.timeout_seconds:
                print(f"Warning: Breaking stale lock from {lock_data.get('hostname')}")
                # Stale lock, break it and try to acquire
                self.lock_path.unlink(missing_ok=True)
                continue

        # Lock is held by another active process, wait and retry
        time.sleep(1)

    raise TimeoutError(f"Could not acquire lock {self.lock_path} within timeout")

def _create_lock_file(self, lock_id: str) -> bool:
    """Atomically create lock file with current process info."""
    try:
        lock_data = {
            'pid': os.getpid(),
            'hostname': os.uname().nodename if hasattr(os, 'uname') else 'unknown',
            'timestamp': time.time(),
            'lock_id': lock_id
        }
        # Use atomic write from serialization module
        from .serialization import write_atomic_json
        write_atomic_json(self.lock_path, lock_data)
        return True
    except (OSError, IOError):
        return False
```

```
def _read_lock_file(self) -> Optional[dict]:
    """Read and parse lock file, return None if invalid/missing."""
    try:
        with open(self.lock_path, 'r') as f:
            return json.load(f)
    except (FileNotFoundError, json.JSONDecodeError):
        return None


def _start_heartbeat(self, handle: LockHandle, interval: int):
    """Start background thread to periodically refresh lock timestamp."""
    def heartbeat():
        while not handle.stop_event.wait(interval):
            if not self.lock_path.exists():
                break
            lock_data = self._read_lock_file()
            if lock_data and lock_data.get('lock_id') == handle.lock_id:
                lock_data['timestamp'] = time.time()
                try:
                    from .serialization import write_atomic_json
                    write_atomic_json(self.lock_path, lock_data)
                except (OSError, IOError):
                    break  # Couldn't refresh, lock may be broken

    handle.heartbeat_thread = threading.Thread(
        target=heartbeat,
        daemon=True,
        name=f"lock-heartbeat-{handle.lock_id[:8]}"
    )
    handle.heartbeat_thread.start()


def release(self, handle: LockHandle):
    """Release the lock represented by the handle."""
    if handle.heartbeat_thread:
        handle.stop_event.set()
        handle.heartbeat_thread.join(timeout=5)

    # Verify we still own the lock before deleting
    lock_data = self._read_lock_file()
```

```
        if lock_data and lock_data.get('lock_id') == handle.lock_id:
            self.lock_path.unlink(missing_ok=True)
```

D. Core Logic Skeleton Code

The main `StateManager` class integrates the above utilities. Learners should implement the `compute_diff` method following the algorithm outlined earlier.

PYTHON

```python
# iac_engine/state/manager.py

"""Main State Manager component."""

import json

from pathlib import Path

from typing import Dict, List, Optional

from dataclasses import asdict


from ..models import Resource, StateRecord, PlanAction, ActionType

from .locking import FileLock, LockHandle

from .serialization import write_atomic_json, read_json_with_backup


class StateManager:

    """Manages persistent infrastructure state and concurrent access."""



    def __init__(self, state_path: Path):

        """

        Args:

            state_path: Path to the main state file (e.g., terraform.tfstate).

        """

        self.state_path = state_path

        self.lock_path = state_path.parent / f"{state_path.name}.lock"

        self.file_lock = FileLock(self.lock_path)



    def read_state(self) -> Dict[str, StateRecord]:

        """

        Loads the state file into memory.



        Returns:

            Dictionary mapping resource addresses to StateRecord objects.

            Returns empty dict if state file doesn't exist or is corrupt.

        """

        # TODO 1: Use read_json_with_backup to load raw JSON from self.state_path

        # TODO 2: If result is empty dict, return empty dict

        # TODO 3: Convert the raw JSON dict into a dict of StateRecord objects

        #         The JSON keys are resource addresses (e.g., "aws_instance.web")

        #         Each value should be converted using StateRecord.from_dict()

        # TODO 4: Return the dictionary of StateRecord objects

        pass



    def write_state(self, state_data: Dict[str, StateRecord]):
```

```
        """
        Persists the state dictionary to disk atomically.

        Args:
            state_data: Dictionary mapping resource addresses to StateRecord objects.
        """
        # TODO 1: Convert the state_data dictionary to a JSON-serializable dict
        #         Use StateRecord.to_dict() for each record
        # TODO 2: Call write_atomic_json with self.state_path and the serialized data
        # TODO 3: (Optional) Create a backup of the previous state file before writing
        pass

    def acquire_lock(self, timeout_seconds: int = 300) -> LockHandle:
        """
        Acquire an exclusive lock for the state.

        Args:
            timeout_seconds: Maximum time to wait for lock acquisition.

        Returns:
            A LockHandle that MUST be passed to release_lock.
        """
        # TODO 1: Call self.file_lock.acquire() with appropriate heartbeat interval
        # TODO 2: Return the LockHandle
        pass

    def release_lock(self, lock_handle: LockHandle):
        """
        Release a previously acquired lock.

        Args:
            lock_handle: The handle returned by acquire_lock.
        """
        # TODO 1: Call self.file_lock.release(lock_handle)
        pass

    def compute_diff(
        self,
        current_state: Dict[str, StateRecord],
```

```
        desired_resources: List[Resource]
    ) -> Dict[str, PlanAction]:
        """
        Compare current state with desired resources to determine needed changes.

        Args:
            current_state: Output from read_state().
            desired_resources: List of Resource objects from the parser.

        Returns:
            Dictionary of PlanAction objects keyed by resource address.
        """
        actions = {}

        # Build index of desired resources by address
        desired_index = {}
        for resource in desired_resources:
            address = self.get_resource_address(resource)
            desired_index[address] = resource

        # TODO 1: Iterate through desired resources (desired_index)
        # TODO 2: For each desired resource, check if it exists in current_state
        # TODO 3: If NOT in current_state -> CREATE action
        #         - prior_state = None
        #         - new_state = StateRecord from resource (resource_id="")
        # TODO 4: If EXISTS in current_state -> compare attributes
        #         - Use _compare_attributes() helper for deep comparison
        #         - If different -> UPDATE action (both states populated)
        #         - If same -> NOOP action
        # TODO 5: After processing desired resources, find resources in current_state
        #         that are NOT in desired_index -> DELETE action
        #         - resource = Resource from StateRecord (type and name)
        #         - prior_state = the StateRecord, new_state = None
        # TODO 6: Add each action to actions dict keyed by resource address
        # TODO 7: Return actions dict

        return actions

    def get_resource_address(self, resource: Resource) -> str:
```

```
        """
        Generate a unique address string for a resource.

        Args:
            resource: A Resource object.

        Returns:
            String in format "{resource.type}.{resource.name}".
        """
        # TODO 1: Return f"{resource.type}.{resource.name}"
        pass

    def _compare_attributes(
        self,
        desired_attrs: Dict[str, any],
        current_attrs: Dict[str, any]
    ) -> bool:
        """
        Deep compare resource attributes, handling provider-specific semantics.

        Args:
            desired_attrs: Attributes from configuration.
            current_attrs: Attributes from cloud API (StateRecord).

        Returns:
            True if attributes are effectively equal, False if changes are needed.
        """
        # TODO 1: Implement a recursive comparison that handles:
        #         - Lists where order may not matter (e.g., security group rules)
        #         - Ignoring read-only/computed fields (like 'arn', 'id')
        #         - Type coercion (string "10" vs integer 10)
        #         - Nested dictionaries
        # TODO 2: Return True if equal, False if any meaningful difference found
        pass
```

E. Language-Specific Hints (Python)

- Use `pathlib.Path` for all file path operations—it's more readable and cross-platform than `os.path`.
- The `json` module's `default=str` parameter in `json.dump` handles non-serializable types (like `datetime`) by converting them to strings.
- For atomic file replacement, `os.replace()` (Python 3.3+) is the most reliable cross-platform method.
- When implementing the heartbeat thread, use `threading.Event` for clean shutdown rather than checking a boolean flag.

- In `_compare_attributes`, consider using a "schema" per resource type to know which fields are computed/read-only and should be ignored during diff.

F. Milestone Checkpoint After implementing the State Manager, verify its functionality:

1. **Test Atomic Writes**: Create a test that writes a state file, simulates a crash (delete temp file mid-write), and ensures the original state remains uncorrupted.
2. **Test Locking**: Run two Python scripts simultaneously that try to acquire the same lock. The second should wait (or timeout) until the first releases it.
3. **Test Diff Logic**: Write unit tests for `compute_diff` with various scenarios: new resource, changed attribute, deleted resource, identical resource (`NOOP`).
4. **Integration Test**: Run `plan_command` on a simple configuration. It should read state (or start with empty), compute a diff, and output a plan showing `CREATE` actions.

Expected Behavior:

- State file is created at `terraform.tfstate` after first successful apply.
- A lock file (`terraform.tfstate.lock`) appears during `plan` / `apply` and disappears afterward.
- Running `plan` twice on the same configuration (without apply) should show the same changes (idempotent diff).

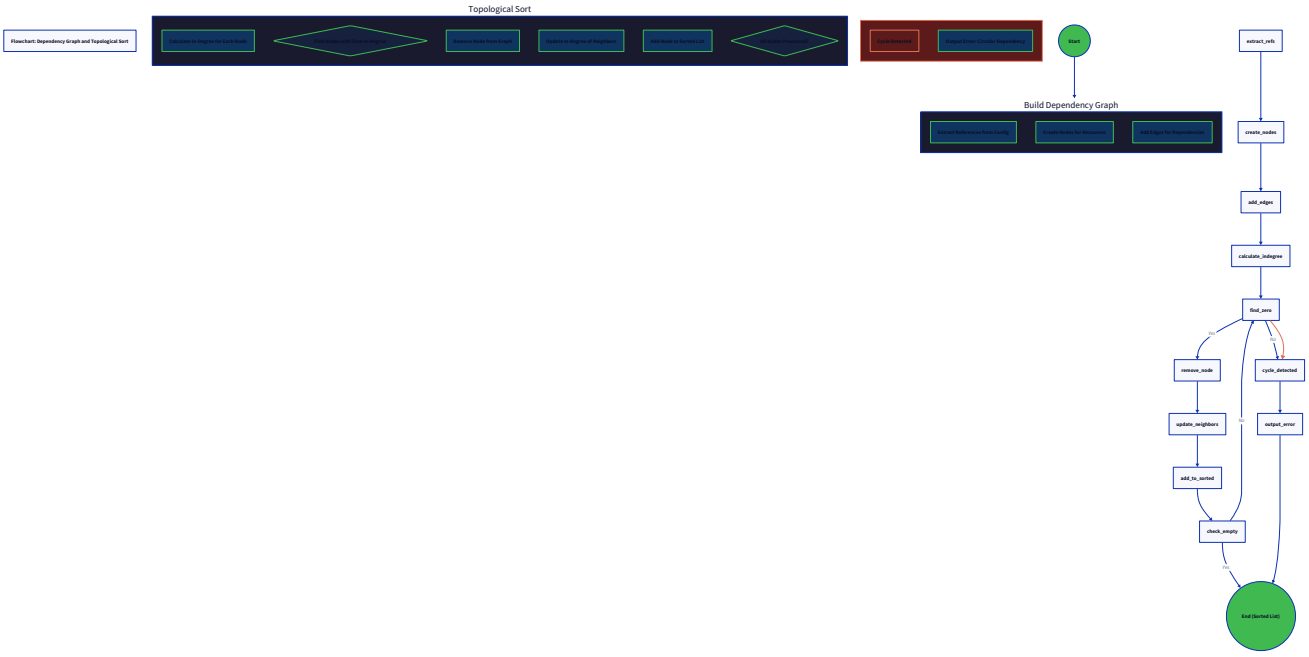
Signs of Trouble:

- "State file is corrupted" errors: Check `write_atomic_json` is being used and temp files are cleaned up.
- "Lock never released": Check heartbeat thread is daemon and `stop_event` is set in `release_lock`. Look for orphaned `.lock` files.
- "Plan shows no changes when resources exist": Verify `compute_diff` attribute comparison logic, especially for nested structures.

Component Design: Planner (Dependency Graph & Planning)

> **Milestone(s):** 3

The **Planner** is the strategic brain of the IaC engine, responsible for transforming a collection of resources (from the parser) and the current state (from the state manager) into a safe, ordered execution plan. It determines **what** needs to change and, critically, **in what order** those changes must be applied to respect dependencies and avoid system failures.



Mental Model: The Project Manager and Gantt Chart

Think of the Planner as a **project manager** preparing to renovate a house. The project manager receives:

1. **The Blueprint:** The desired configuration from the parser (like the architect's new plans).
2. **The As-Built Drawings:** The current state from the state manager (what's actually constructed today).

The manager's job is to compare these two documents and create a **detailed work plan (execution plan)** for the construction crew. To do this, they must:

- **Identify Task Dependencies (Build the Dependency Graph):** They analyze which tasks depend on others. You cannot install light fixtures until the electrical wiring is run. You cannot pour the foundation until the site is excavated. These dependencies form a **Directed Acyclic Graph (DAG)** of tasks. A cycle (e.g., "Task A needs B, but B also needs A") indicates a planning error and must be caught.
- **Sequence the Work (Topological Sort):** Using the DAG, the manager creates a **Gantt chart** – a linear sequence of tasks where all dependencies are satisfied before a task begins. This is the topological sort.
- **Determine Action Types (Compute the Diff):** For each task (resource), the manager decides if it's a **new addition (CREATE)**, a **modification (UPDATE)**, something to be **removed (DELETE)**, or if it's already correct and needs **no action (NOOP)**.
- **Plan for Teardown (Reverse Destroy Order):** If removing parts of the house, the manager knows to work in reverse: remove light fixtures *before* removing the wiring, and remove the wiring *before* removing the wall it's in. The Planner ensures **destroy operations follow the reverse dependency order**.

This mental model emphasizes the Planner's core responsibilities: **dependency analysis, ordering, and change calculation** to produce a safe, actionable plan.

Interface and Public Methods

The Planner component exposes a clean interface focused on graph construction and plan generation. Its primary client is the CLI layer when executing the `plan_command`.

Method Name	Parameters	Returns	Description
`build_graph`	`resources: List[Resource]`, `state_records: Dict[str, StateRecord]`	`Dict[str, DependencyGraphNode]`	Constructs a dependency graph from the provided resources and existing state. Extracts both implicit (attribute reference) and explicit (`depends_on`) dependencies. Returns a dictionary mapping resource addresses to their graph node.
`validate_acyclic`	`graph: Dict[str, DependencyGraphNode]`	`bool`	Validates that the dependency graph contains no cycles. Returns `True` if the graph is acyclic; raises a `CycleDetectionError` with details of the detected cycle if not.
`topological_sort`	`graph: Dict[str, DependencyGraphNode]`	`List[str]`	Performs a topological sort on the validated DAG. Returns an ordered list of resource addresses where each resource appears **after** all its dependencies.
`generate_plan`	`sorted_resources: List[str]`, `desired_resources: Dict[str, Resource]`, `current_state: Dict[str, StateRecord]`	`Dict[str, PlanAction]`	The core planning algorithm. Iterates through resources in topological order, comparing each desired `Resource` against its corresponding `StateRecord` to produce a `PlanAction`. Returns a dictionary of actions keyed by resource address.
`plan_command` *(CLI)*	`config_path: Path`, `state_path: Path`, `var_file: Optional[Path]`	`None`	The orchestrating CLI command. Calls the parser (`process_configuration`), reads state (`read_state`), calls the planner's methods (`build_graph` , `validate_acyclic` , `topological_sort` , `generate_plan`), and prints a human-readable summary of the execution plan.

Internal Behavior and Algorithm

The Planner's internal logic is a multi-stage pipeline. Each stage transforms data, progressively refining the raw configuration and state into a precise execution plan.

1. Dependency Graph Construction (`build_graph`)

This algorithm builds the `DependencyGraphNode` for each resource. A `DependencyGraphNode` tracks what a resource **depends on** (`depends_on`) and what **depends on it** (`required_by`).

Field	Type	Description
`resource_id`	`str`	The unique address of the resource (e.g., `aws_instance.web_server`).
`depends_on`	`List[str]`	List of resource addresses that **this resource** depends on. Populated during dependency extraction.
`required_by`	`List[str]`	List of resource addresses that depend on **this resource**. Populated as edges are added, creating the reverse link for efficient traversal.

Algorithm Steps:

1. **Initialize Graph:** Create an empty dictionary `graph: Dict[str, DependencyGraphNode]`. For each resource in `resources` and each existing resource in `state_records`, create a node with empty `depends_on` and `required_by` lists.
2. **Extract Explicit Dependencies:** For each resource, check for an explicit `depends_on` attribute in its `attributes` map. If present, validate that each referenced resource address exists in the combined set of desired and existing resources. For each valid reference, add the dependency to the current resource's `depends_on` list.
3. **Extract Implicit Dependencies:** This is the most complex step. For each resource, recursively scan all values in its `attributes` map (including nested maps and lists) for **interpolation references**. A reference pattern is `${<reference>}`. We are interested in references to other resources, which typically follow patterns like `${aws_security_group.my_sg.id}` or `${module.vpc.subnet_id}`.
   - **Parse the Reference:** Extract the referenced path (e.g., `aws_security_group.my_sg.id`).
   - **Resolve to Resource Address:** The core of the path before the first dot (`.`) or opening bracket (`[`) is the **resource address** (e.g., `aws_security_group.my_sg`). You may need to handle module outputs (e.g., `module.vpc.subnet_id` resolves to a resource inside the `vpc` module). The Planner relies on the Parser having already **flattened** module resources and resolved their addresses to global, unique strings.
   - **Add Dependency Edge:** If the resolved resource address exists in the graph, add an edge from the current resource (dependent) to the referenced resource (dependency). This means adding the dependency's address to the current resource's `depends_on` list and adding the current resource's address to the dependency's `required_by` list.
4. **Return Graph:** Return the completed `graph` dictionary.

2. Cycle Detection (`validate_acyclic`)

A cycle in the dependency graph means no valid execution order exists (e.g., Resource A needs B created first, but B needs A created first). We must detect and report this before proceeding.

Algorithm Steps (Kahn's Algorithm or DFS):

1. Use **Depth-First Search (DFS)** with coloring:
   - `WHITE` = Unvisited node
   - `GREY` = Currently visiting (in the DFS stack)
   - `BLACK` = Fully processed
2. For each node in the graph, if it's `WHITE`, start a DFS.
3. When entering a node, mark it `GREY`.
4. For each dependency in the node's `depends_on` list:
   - If the dependency node is `GREY`, a cycle has been found. Construct an error message tracing the cycle path and raise a `CycleDetectionError`.
   - If the dependency node is `WHITE`, recursively visit it.
5. After processing all dependencies, mark the node `BLACK`.
6. If the DFS completes for all nodes without finding a `GREY` dependency, the graph is acyclic.

3. Topological Sort (`topological_sort`)

Given an acyclic graph, we produce a linear execution order. **Kahn's Algorithm** is intuitive and efficient.

Algorithm Steps:

1. **Compute In-Degree:** For each node in the graph, compute its "in-degree" – the count of resources that depend on it (the length of its `required_by` list). You can derive this from the `graph` structure.
2. **Initialize Queue:** Create a queue (or list) and add all nodes with an in-degree of `0`. These are resources with no dependencies and can be executed first.
3. **Process Queue:** While the queue is not empty: a. Remove a node `n` from the queue. b. Append `n.resource_id` to the result list `sorted_order`. c. For each node `m` that `n` depends on (i.e., for each node in `n.depends_on`), decrement the in-degree of `m` by `1`. d. If the in-degree of `m` becomes `0`, add `m` to the queue.
4. **Check for Cycles (Again):** After the queue is empty, if the length of `sorted_order` is less than the total number of nodes in the graph, it indicates a cycle (though `validate_acyclic` should have caught this). This is a final sanity check.
5. **Return Order:** Return the `sorted_order` list.

4. Plan Generation (`generate_plan`)

This function translates the desired state, current state, and execution order into concrete `PlanAction` objects. It must handle the four fundamental action types.

Field	Type	Description
`action_type`	`ActionType` (Enum)	One of: `CREATE`, `UPDATE`, `DELETE`, `NOOP`.
`resource`	`Resource`	The desired `Resource` object. For `DELETE` actions, this is the resource *as it exists in the current state* (or a placeholder).
`prior_state`	`Optional[StateRecord]`	The state of the resource before the action. Present for `UPDATE` and `DELETE`; `None` for `CREATE`.
`new_state`	`Optional[StateRecord]`	The expected state of the resource after the action. Present for `CREATE` and `UPDATE`; `None` for `DELETE`.

Algorithm Steps:

1. **Initialize Plan:** Create an empty dictionary `plan: Dict[str, PlanAction]`.
2. **Iterate in Sorted Order:** Loop through each `resource_id` in the `sorted_resources` list.
3. **Retrieve States:** Get the `desired_resource` from `desired_resources` dict and the `current_record` from `current_state` dict for this `resource_id`. One may be `None`.
4. **Determine Action Type:**
   - If `desired_resource` exists and `current_record` is `None` → `ActionType.CREATE`
   - If `desired_resource` exists and `current_record` exists → Compare their `attributes`. If any attribute differs (deep comparison), → `ActionType.UPDATE`. If identical → `ActionType.NOOP`.
   - If `desired_resource` is `None` and `current_record` exists → `ActionType.DELETE`
5. **Construct PlanAction:**
   - For `CREATE`: `action_type=CREATE`, `resource=desired_resource`, `prior_state=None`, `new_state=a StateRecord derived from desired_resource`.
   - For `UPDATE`: `action_type=UPDATE`, `resource=desired_resource`, `prior_state=current_record`, `new_state=a StateRecord derived from desired_resource`.
   - For `DELETE`: `action_type=DELETE`, `resource=a Resource constructed from current_record`, `prior_state=current_record`, `new_state=None`.
   - For `NOOP`: `action_type=NOOP`, `resource=desired_resource`, `prior_state=current_record`, `new_state=current_record`.
6. **Store Action:** Add the `PlanAction` to the `plan` dictionary keyed by `resource_id`.
7. **Return Plan:** After processing all resources, return the `plan`.

> **Key Design Insight:** The plan is generated **in dependency order**, but the `PlanAction` objects themselves are stored in a dictionary for easy lookup by the Executor. The Executor will later use the same topological order (or its reverse for destroy) to sequence operations.

ADR: Implicit vs. Explicit Dependency Detection

Decision: Implement Both Implicit and Explicit Dependency Detection

- **Context:** Resources in an IaC configuration can depend on each other in two ways: 1) **Explicitly** via a `depends_on` directive, and 2) **Implicitly** via attribute references (e.g., using another resource's ID in a configuration string). We must decide which types of dependencies the Planner will recognize to build an accurate and safe execution graph.
- **Options Considered:**
  1. **Explicit-Only Detection:** Only process the `depends_on` attribute. This is simple and fast but fails to catch the majority of real-world dependencies expressed through interpolation, leading to runtime failures when a resource references an attribute of another resource that hasn't been created yet.
  2. **Implicit-Only Detection:** Only parse attribute values for interpolation references. This catches the most common dependency patterns but forces users to understand the engine's reference parsing logic. It may also miss dependencies expressed through complex, indirect means not caught by the parser.
  3. **Hybrid Detection (Explicit + Implicit):** Process both `depends_on` and interpolated references. This provides robustness and user flexibility. Explicit `depends_on` can be used to force an ordering when implicit detection fails or is unclear.
- **Decision:** Implement **Hybrid Detection (Option 3)**. The Planner will scan for and process both explicit `depends_on` directives and implicit references found in attribute values.
- **Rationale:** The primary goal of the Planner is to guarantee a safe apply order. Relying solely on explicit dependencies places too much burden on the user and is error-prone. Relying solely on implicit detection is complex and may have blind spots. The hybrid approach offers a safety net: implicit detection handles the common cases automatically, while explicit `depends_on` allows users to override or clarify dependencies when necessary (e.g., for resources managed by a provider that don't expose useful attributes, or for meta-dependencies not expressed in configuration).
- **Consequences:**
  - **Increased Complexity:** The dependency extraction logic must handle two distinct patterns and resolve interpolated strings to resource addresses. This requires a well-defined reference syntax and a robust parsing step.
  - **Potential for Over-Specification:** Users might add unnecessary `depends_on` statements, which clutter the config but do not cause harm.
  - **Accurate Graphs:** The resulting dependency graph more closely reflects the true operational dependencies of the infrastructure, leading to more reliable `apply` operations.

Option	Pros	Cons	Chosen?
Explicit-Only	Simple to implement and understand.	Fragile; misses most real dependencies, causing apply failures.	❌
Implicit-Only	Catches common patterns automatically; reduces config clutter.	Complex to implement; may miss edge cases; no user override.	❌
Hybrid	Robust; provides safety net with user override; mirrors industry standards (Terraform).	More complex implementation; requires parsing interpolation.	✅

Common Pitfalls

⚠️ Pitfall: Uncaught Circular Dependencies

- **Description:** A configuration where Resource A depends on B, and B depends on A (directly or indirectly) creates a cycle. If the cycle detection algorithm fails (e.g., only checks explicit `depends_on` but not implicit references), the topological sort will either fail silently or produce an invalid order.
- **Why it's Wrong:** Applying such a plan is impossible. The engine would be stuck trying to create A before B and B before A. In practice, this can lead to infinite loops, partial applies, or corrupted state.
- **How to Fix:** Implement robust cycle detection using DFS coloring on the **full hybrid graph**. When a cycle is detected, **fail fast** with a clear error message listing the resources involved in the cycle. Do not attempt to generate a plan.

⚠️ Pitfall: Missed Implicit Dependencies from Complex Interpolation

- **Description:** The implicit reference scanner only looks for simple `${type.name.attr}` patterns. It might miss dependencies hidden within string concatenation (`"prefix-${var.sg_id}"`), complex functions (`md5("${aws_instance.a.id}-salt")`), or conditional logic (`count = var.create ? 1 : 0`).
- **Why it's Wrong:** The execution plan will be missing critical edges. During apply, a resource may try to reference an attribute from another resource that hasn't been created yet, causing a provider API error.
- **How to Fix:** 1) Document the limitation and encourage use of explicit `depends_on` for complex cases. 2) Implement a more sophisticated expression evaluator in the Parser that can resolve variable and resource references *statically* and export a list of referenced resources for the Planner. For Milestone

3, a best-effort scanner for common patterns is acceptable, with `depends_on` as the escape hatch.

⚠️ **Pitfall: Destroy Operations in Incorrect Order**

- **Description:** Applying a plan that deletes resources using the same order as creation (topological sort). This tries to delete a resource (e.g., a database instance) before deleting resources that depend on it (e.g., a firewall rule pointing to that instance), which will fail because the provider API will reject the deletion of the dependent resource.
- **Why it's Wrong:** Destruction must respect dependency order in **reverse**. The dependent resource must be removed before the resource it depends on.
- **How to Fix:** When generating a plan that contains `DELETE` actions, the Executor (covered in the next section) must **reverse the topological order** for the destroy portion of the plan. The Planner's `topological_sort` provides the create/update order. The reverse of that order is the safe destroy order.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option
Graph Data Structure	Python dictionaries and lists: `Dict[str, DependencyGraphNode]`. Use `List[str]` for `depends_on` and `required_by`.	NetworkX library for professional-grade graph algorithms and visualization.
Cycle Detection & Topological Sort	Implement Kahn's Algorithm or DFS manually (good learning exercise).	Use `networkx.is_directed_acyclic_graph` and `networkx.topological_sort`.
Deep Comparison for Diffs	Python's `==` operator with `json.dumps` serialization or recursive dictionary comparison.	Use a library like `deepdiff` for detailed diff reporting.

B. Recommended File/Module Structure

Add the Planner component to the established project layout.

```
iac_engine/
├── cmd/
│   └── cli.py              # CLI commands: `plan_command`, `apply_command`
├── core/                   # Core data types and interfaces
│   ├── __init__.py
│   ├── models.py           # Resource, StateRecord, PlanAction, DependencyGraphNode, ActionType Enum
│   └── provider.py         # BaseProvider interface
├── parser/                 # Milestone 1
│   └── ...
├── state/                  # Milestone 2
│   └── ...
├── planner/                # Milestone 3 (THIS COMPONENT)
│   ├── __init__.py
│   ├── builder.py          # build_graph, dependency extraction logic
│   ├── graph_utils.py      # validate_acyclic, topological_sort, CycleDetectionError
│   ├── plan_generator.py   # generate_plan
│   └── errors.py           # Planner-specific exceptions
├── executor/               # Milestone 4
│   └── ...
└── providers/              # Milestone 4
    └── ...
```

C. Infrastructure Starter Code

Complete `DependencyGraphNode` and `CycleDetectionError`:

```python
# File: core/models.py

from typing import Dict, List, Optional, Any

from dataclasses import dataclass, field

from enum import Enum


# ... (existing Resource, StateRecord, PlanAction definitions) ...


class ActionType(Enum):

    CREATE = "create"

    UPDATE = "update"

    DELETE = "delete"

    NOOP = "noop"


@dataclass

class DependencyGraphNode:

    """Represents a node in the resource dependency DAG."""

    resource_id: str

    depends_on: List[str] = field(default_factory=list)

    required_by: List[str] = field(default_factory=list)


    def add_dependency(self, dep_id: str) -> None:

        """Adds a dependency edge from this node to dep_id."""

        if dep_id not in self.depends_on:

            self.depends_on.append(dep_id)
```

```python
# File: planner/errors.py

class CycleDetectionError(Exception):

    """Raised when a circular dependency is detected in the resource graph."""

    def __init__(self, cycle_path: List[str]):

        self.cycle_path = cycle_path

        message = f"Circular dependency detected: {' -> '.join(cycle_path)}"

        super().__init__(message)
```

Generic Graph Utilities Skeleton:

PYTHON

```python
# File: planner/graph_utils.py

from typing import Dict, List

from .errors import CycleDetectionError

from core.models import DependencyGraphNode


def validate_acyclic(graph: Dict[str, DependencyGraphNode]) -> bool:

    """

    Validates that the graph contains no cycles using DFS with coloring.

    Returns True if acyclic, raises CycleDetectionError otherwise.

    """

    WHITE, GREY, BLACK = 0, 1, 2

    color: Dict[str, int] = {node_id: WHITE for node_id in graph}

    stack: List[str] = []



    def dfs(node_id: str) -> None:

        color[node_id] = GREY

        stack.append(node_id)



        for dep_id in graph[node_id].depends_on:

            if dep_id not in color:

                # Dependency not in graph (maybe from a module not included). Ignore or handle.

                continue

            if color[dep_id] == GREY:

                # Cycle found! The cycle is from dep_id to the node_id in the stack.

                cycle_start = stack.index(dep_id)

                cycle = stack[cycle_start:] + [dep_id]

                raise CycleDetectionError(cycle)

            if color[dep_id] == WHITE:

                dfs(dep_id)



        stack.pop()

        color[node_id] = BLACK



    for node_id in graph:

        if color[node_id] == WHITE:

            dfs(node_id)

    return True


def topological_sort(graph: Dict[str, DependencyGraphNode]) -> List[str]:

    """
```

```
    Performs a topological sort on a validated DAG using Kahn's Algorithm.
    Returns a list of resource IDs in execution order.
    """
    from collections import deque

    # 1. Compute in-degree for each node
    in_degree: Dict[str, int] = {node_id: 0 for node_id in graph}
    for node in graph.values():
        for dep_id in node.depends_on:
            if dep_id in in_degree:  # Only count dependencies present in the graph
                in_degree[dep_id] += 1

    # 2. Initialize queue with nodes having zero in-degree
    queue = deque([node_id for node_id, deg in in_degree.items() if deg == 0])
    sorted_order: List[str] = []

    # 3. Process queue
    while queue:
        node_id = queue.popleft()
        sorted_order.append(node_id)

        for dep_id in graph[node_id].depends_on:
            if dep_id not in in_degree:
                continue
            in_degree[dep_id] -= 1
            if in_degree[dep_id] == 0:
                queue.append(dep_id)

    # 4. Check for cycles (should not happen if validate_acyclic was called)
    if len(sorted_order) != len(graph):
        missing = set(graph.keys()) - set(sorted_order)
        raise CycleDetectionError(f"Graph has cycle(s) involving: {missing}")

    return sorted_order
```

D. Core Logic Skeleton Code

Dependency Graph Builder:

PYTHON

```python
# File: planner/builder.py

import re

from typing import Dict, List, Any

from pathlib import Path

from core.models import Resource, StateRecord, DependencyGraphNode


# Simple pattern to find interpolation references: ${...}

# This is a simplified version. A real implementation would need a proper HCL expression parser.

REF_PATTERN = re.compile(r'\$\{([^}]+)\}')


def extract_resource_address_from_ref(ref: str) -> str:

    """

    Attempts to extract a resource address from an interpolation reference.

    Example: 'aws_security_group.my_sg.id' -> 'aws_security_group.my_sg'

             'module.vpc.subnet_id' -> 'module.vpc' (module resources are flattened by parser)

    This is a heuristic and may need refinement.

    """

    # Split by '.' and take the first two parts if it looks like a resource reference.

    parts = ref.split('.')

    if len(parts) >= 2:

        # Check if the first part is a resource type or 'module'

        if parts[0] in ('module', 'data') or not parts[0].startswith('var'):

            return f"{parts[0]}.{parts[1]}"

    # If we can't parse it, return the original ref. The caller will filter it out.

    return ref


def find_references_in_value(value: Any) -> List[str]:

    """

    Recursively traverses a value (which can be dict, list, str, etc.)

    and extracts all interpolation reference strings.

    """

    refs = []

    if isinstance(value, str):

        matches = REF_PATTERN.findall(value)

        refs.extend(matches)

    elif isinstance(value, dict):

        for v in value.values():

            refs.extend(find_references_in_value(v))

    elif isinstance(value, list):

        for item in value:
```

```
            refs.extend(find_references_in_value(item))
    return refs

def build_graph(resources: List[Resource],
                state_records: Dict[str, StateRecord]) -> Dict[str, DependencyGraphNode]:
    """
    Builds a dependency graph from desired resources and existing state records.
    """
    graph: Dict[str, DependencyGraphNode] = {}

    # TODO 1: Create nodes for all resources and existing state records.
    #   For each resource in `resources`, create a DependencyGraphNode with resource_id = get_resource_address(resource).
    #   For each record in `state_records`, also create a node if one doesn't already exist.
    #   Add all nodes to the `graph` dict keyed by resource_id.

    # TODO 2: Process explicit dependencies (depends_on).
    #   For each resource in `resources`:
    #       Check its attributes for a key named 'depends_on'. The value is likely a list of strings.
    #       For each dependency string in that list:
    #           Validate the dependency exists in the `graph`.
    #           Add the dependency to the current resource's node.depends_on list.
    #           Add the current resource to the dependency node's required_by list.

    # TODO 3: Process implicit dependencies (interpolation references).
    #   For each resource in `resources`:
    #       Recursively find all interpolation references in its attributes using `find_references_in_value`.
    #       For each reference found:
    #           Attempt to resolve it to a resource address using `extract_resource_address_from_ref`.
    #           If the resolved address exists in the `graph` and is NOT the resource itself:
    #               Add an implicit dependency edge (same as step 2).

    # TODO 4: Return the completed graph.
    return graph
```

Plan Generator:

```python
# File: planner/plan_generator.py

from typing import Dict, List

from core.models import Resource, StateRecord, PlanAction, ActionType, get_resource_address


def generate_plan(sorted_resource_ids: List[str],

                  desired_resources: Dict[str, Resource],

                  current_state: Dict[str, StateRecord]) -> Dict[str, PlanAction]:

    """

    Generates a PlanAction for each resource based on desired vs current state.

    """

    plan: Dict[str, PlanAction] = {}



    # TODO 1: Iterate through each resource_id in the sorted_resource_ids list.



    # TODO 2: For each resource_id, retrieve:

    #   desired = desired_resources.get(resource_id)

    #   current = current_state.get(resource_id)



    # TODO 3: Determine the action type based on the presence of desired and current.

    #   Use the logic described in the "Plan Generation" algorithm.

    #   For UPDATE, you need to compare attributes. A simple way is:

    #       import json; is_update = json.dumps(desired.attributes) != json.dumps(current.attributes)



    # TODO 4: Construct the appropriate PlanAction object.

    #   Remember to set prior_state and new_state correctly for each action type.

    #   For CREATE, new_state should be a StateRecord created from the desired Resource.

    #   For DELETE, the PlanAction's 'resource' field should be derived from the current_state.



    # TODO 5: Add the PlanAction to the plan dictionary with resource_id as the key.



    # TODO 6: After the loop, return the plan dictionary.

    return plan
```

E. Language-Specific Hints

- Use Python's `@dataclass` decorator from the `dataclasses` module for clean model definitions like `DependencyGraphNode`. It automatically generates `__init__`, `__repr__`, and `__eq__` methods.
- For recursive search in `find_references_in_value`, using `isinstance()` checks for `dict`, `list`, and `str` is a straightforward approach. Be mindful of recursion depth for very large configurations.
- When comparing attribute dictionaries for updates, serializing to JSON with `json.dumps` and comparing strings is a simple, order-insensitive way to perform a deep equality check. For more complex comparisons (e.g., ignoring certain fields), consider using the `deepdiff` library.
- The `re` module is sufficient for basic interpolation pattern matching. For a production-grade parser, you would integrate with the actual expression AST from the Parser component.

Component Design: Provider Abstraction & Executor

> **Milestone(s):** 4

The **Provider Abstraction & Executor** represents the "muscle" of the IaC engine. While previous components understand *what* needs to change, this component actually *makes* those changes by interacting with external cloud platforms through a clean plugin architecture. The **Executor** orchestrates the safe application of the execution plan, while the **Provider Abstraction** defines the uniform interface that all cloud-specific implementations must follow.

Mental Model: The Universal Remote Control

Imagine you have a universal remote control that can operate any brand of TV. The remote has a standard set of buttons: Power, Volume Up/Down, Channel Change. However, each TV manufacturer requires different infrared codes to execute these same actions. The universal remote solves this by having a collection of "device codes" – each code translates the standard button press into the specific signals that a particular TV model understands.

In this analogy:

- The **`BaseProvider` interface** is the standard set of buttons (CRUD operations) that every remote must have.
- Each **provider implementation** (AWS, Azure, GCP) is a specific device code that translates those standard operations into the exact HTTP API calls, authentication mechanisms, and error handling for that cloud.
- The **Executor** is the person holding the remote, who knows *when* to press which buttons and in what order, based on the execution plan (the "channel guide").

This separation achieves **polymorphism**: the Executor can orchestrate infrastructure changes without knowing the details of any specific cloud API. When we need to support a new cloud, we simply add a new "device code" (provider implementation) without modifying the orchestration logic.

Interface and Public Methods

The Provider Abstraction is defined by a Python abstract base class (ABC) that all concrete providers must implement. The Executor is a separate class that consumes this interface to apply changes.

BaseProvider Interface

All providers must implement these CRUD lifecycle methods plus configuration validation.

Method	Parameters	Returns	Description
`create`	`resource: Resource`	`Resource`	Creates a new cloud resource. Takes a `Resource` with desired attributes, calls the cloud API, and returns a `Resource` populated with the actual created resource's attributes (including its cloud-assigned ID). Must be idempotent.
`read`	`resource_id: str, resource_type: str`	`Optional[Resource]`	Reads the current state of a resource from the cloud API using its unique identifier. Returns a `Resource` if the resource exists, or `None` if it does not. Used for state refresh and validation.
`update`	`resource_id: str, resource: Resource`	`Resource`	Updates an existing cloud resource. Compares current attributes (from `read`) with desired attributes, makes necessary API calls, and returns the updated `Resource`. Must handle partial updates and be idempotent.
`delete`	`resource_id: str, resource_type: str`	`bool`	Deletes a cloud resource. Returns `True` if deletion succeeded or the resource was already absent, `False` on unrecoverable failure. Must be idempotent (deleting a non-existent resource is a success).
`validate_credentials`	`config: dict`	`bool`	Validates that the provider configuration (credentials, region, etc.) is correct and can authenticate with the cloud API. Returns `True` on success, raises a descriptive exception on failure.
`get_schema`	`resource_type: str`	`dict`	(Optional) Returns the attribute schema for a given resource type, used for validation. Includes required vs. optional fields, data types, and allowed values.

Executor Class

The Executor applies a plan by calling provider methods in the correct order, with safety mechanisms.

Method	Parameters	Returns	Description
`apply_plan`	`plan: Dict[str, PlanAction], providers: Dict[str, BaseProvider], max_concurrency: int = 1`	`Dict[str, ApplyResult]`	Main entry point. Applies the execution plan using the provided provider instances. Respects dependency ordering (plan already topologically sorted). Supports limited concurrency for independent resources. Returns a dictionary mapping resource addresses to `ApplyResult` objects (success/failure, new state, errors).
`refresh_state`	`resource: Resource, provider: BaseProvider`	`Optional[Resource]`	Helper method that calls `provider.read` to get the current cloud state of a single resource. Used before updates and after creates to populate the state file with accurate attributes.
`_execute_action`	`action: PlanAction, provider: BaseProvider`	`ApplyResult`	Internal method that executes a single `PlanAction` (CREATE, UPDATE, DELETE) by calling the corresponding provider method. Wraps the call with retry logic, timeout, and error handling.

Supporting Data Structures:

Type Name	Fields	Description
`ApplyResult`	`resource_address: str`, `success: bool`, `new_state: Optional[Resource]`, `error: Optional[str]`, `retries: int`	Captures the outcome of applying a single `PlanAction`. Used to report success/failure back to the Executor and ultimately the user.
`ProviderConfig`	`provider_type: str`, `config: dict`	Holder for provider-specific configuration (e.g., AWS region, access keys). Loaded from configuration files and passed to provider factory functions.

Internal Behavior and Algorithm

The Executor's primary algorithm is the `apply_plan` method, which transforms a set of planned actions into actual cloud resources. The algorithm must be robust to partial failures, API rate limits, and eventual consistency.

Algorithm: Executing an Infrastructure Plan

1. **Input Validation & Setup**
   - Verify that for every `PlanAction` in the plan, a corresponding `BaseProvider` instance exists in the `providers` dictionary (keyed by provider type, e.g., "aws").
   - Initialize an empty results dictionary to track `ApplyResult` for each resource.
   - If `max_concurrency > 1`, initialize a thread pool or async task queue. For simplicity, our reference implementation will use sequential execution (`max_concurrency = 1`).
2. **Sequential Execution Following Topological Order**
   - The `plan` dictionary is assumed to be already ordered by the Planner via topological sort (keys are resource addresses in execution order).
   - For each resource address in the sorted order: a. Retrieve the `PlanAction` for this resource. b. Retrieve the appropriate `BaseProvider` instance based on the resource's type (e.g., "aws_instance" → "aws" provider). c. Call `_execute_action(action, provider)`. d. Store the resulting `ApplyResult` in the results dictionary. e. **Critical: Update In-Memory State Immediately.** If the action succeeded, update an in-memory copy of the state with the `new_state` from the result. This ensures subsequent actions that depend on this resource see the updated state (e.g., a created resource's ID for reference).
3. **Single Action Execution (`_execute_action`)** For a given `PlanAction` and `BaseProvider`:
   1. Determine the operation from `action.action_type`:
      - `ActionType.CREATE`:
        - Call `provider.create(action.resource)` with retry logic (see ADR below).
        - After successful creation, call `provider.read` to refresh and capture all server-generated attributes (e.g., cloud IDs, timestamps).
        - Return an `ApplyResult` with `success=True` and the refreshed resource as `new_state`.
      - `ActionType.UPDATE`:
        - First, call `provider.read` to get the current actual state from the cloud.
        - Compare attributes with `action.resource` to compute the minimal update delta (some providers require full replacement).
        - Call `provider.update(resource_id, action.resource)` with retry logic.
        - Call `provider.read` again to refresh state.

        - Return `ApplyResult` with the refreshed resource.
    - `ActionType.DELETE`:
        - Call `provider.delete(resource_id, resource_type)` with retry logic.
        - Verify deletion by calling `provider.read`; it should return `None`.
        - Return `ApplyResult` with `success=True` and `new_state=None`.
    - `ActionType.NOOP`: Return a successful `ApplyResult` with the existing state.
2. **Retry & Error Handling Wrapper:**
    - Wrap each provider call in a retry decorator implementing exponential backoff with jitter.
    - Catch transient errors (network timeouts, rate limit exceedances, 5xx status codes) and retry.
    - On permanent errors (invalid credentials, unsupported resource type, validation errors), fail immediately and return an `ApplyResult` with `success=False` and the error message.
3. **Timeout Protection:**
    - Each provider call should have a configurable timeout (e.g., 300 seconds for long operations).
    - If a timeout occurs, treat it as a transient error and retry (up to the retry limit).
4. **Post-Application State Persistence**
    - After all actions are completed (or upon early failure if we decide to stop), the Executor returns the results dictionary to the caller (typically the CLI).
    - The caller (CLI) is responsible for writing the updated in-memory state to the state file using `write_state`, but only if the overall apply was successful (or partially successful with a flag to save). This ensures the state file only reflects changes that actually occurred.

> **Key Insight:** The Executor must update its internal state representation *immediately* after each successful action, not just at the very end. This is because subsequent resources in the dependency order may reference attributes (like an ID) of a resource just created. If we don't update the internal view, those references would be pointing to stale or non-existent data.

State Machine: Resource Lifecycle During Apply

The following table describes the states a resource transitions through during the `apply_plan` operation, corresponding to the

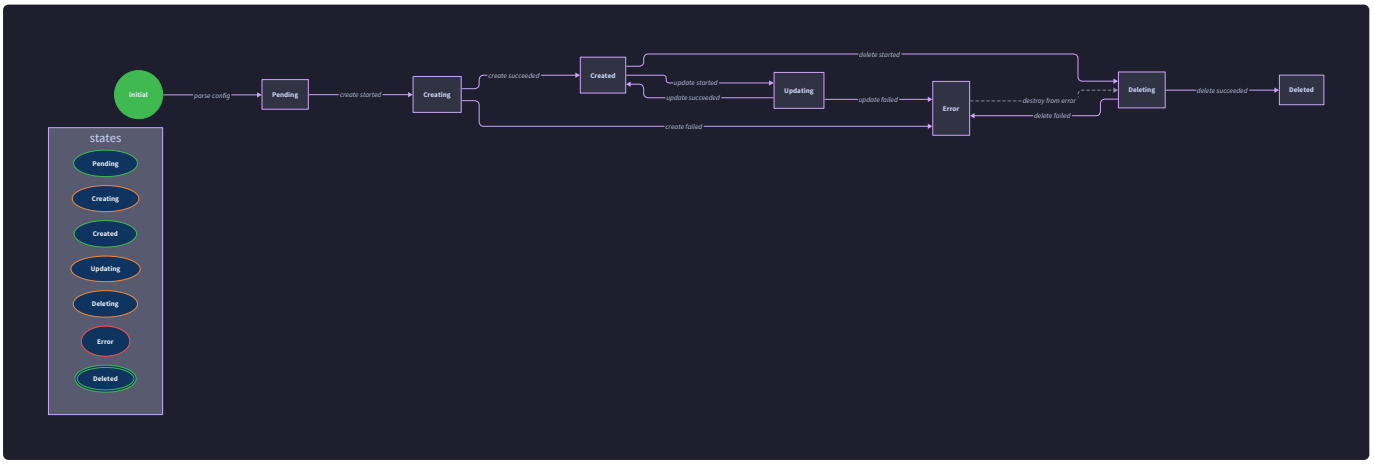


diagram.

Current State	Event	Next State	Action Taken
Pending	`execute_action` called (CREATE)	Creating	Call `provider.create` with retry wrapper; start timeout timer.
Creating	`provider.create` succeeds	Created	Call `provider.read` to refresh; store new state in memory; mark success.
Creating	`provider.create` fails (permanent error)	Error	Store error in `ApplyResult`; do not update in-memory state.
Creating	Timeout or transient error (with retries exhausted)	Error	Store timeout/error; do not update state.
Pending	`execute_action` called (UPDATE)	Updating	Call `provider.read` → `provider.update` with retry wrapper.
Updating	`provider.update` succeeds	Created	Call `provider.read` to refresh; update in-memory state.
Updating	`provider.update` fails or timeout	Error	Store error; keep prior state in memory.
Pending	`execute_action` called (DELETE)	Deleting	Call `provider.delete` with retry wrapper; verify with `provider.read`.
Deleting	`provider.delete` succeeds (resource gone)	Deleted	Remove resource from in-memory state; mark success.
Deleting	`provider.delete` fails (permanent)	Error	Store error; resource remains in state (manual intervention needed).
Any state	User interrupt (Ctrl+C)	Interrupted	Attempt graceful cancellation of ongoing API call; mark as interrupted.

ADR: Retry Logic Strategy

> **Decision: Exponential Backoff with Jitter for Transient Failures**

- **Context**: Cloud APIs are distributed systems that experience transient failures: network timeouts, rate limiting (429), internal server errors (5xx), and eventual consistency delays. The IaC engine must be resilient to these temporary issues without requiring user intervention, while also avoiding retry storms that exacerbate problems.
- **Options Considered**:
  1. **No retries** – Fail immediately on any error. Simplest to implement but not resilient, leading to frequent apply failures from transient issues.
  2. **Fixed-interval retries** – Retry a fixed number of times with a constant delay (e.g., 5 seconds). Better than no retries, but can create synchronized retry bursts that overwhelm recovering services.
  3. **Exponential backoff with jitter** – Increase wait times exponentially between retries (e.g., 1s, 2s, 4s, 8s) and add random variation (jitter) to spread retry attempts across time.
- **Decision**: Implement **exponential backoff with jitter** as the default retry strategy for all provider API calls.
- **Rationale**:
  - Exponential backoff respects the cloud provider's potential overload by progressively reducing retry pressure, aligning with the TCP congestion control principle.
  - Jitter (randomization) prevents thundering herd problems where many failed requests all retry at the same time, which is especially important in multi-tenant or parallel apply scenarios.
  - This pattern is a well-established best practice for distributed system clients (AWS SDKs, Google Cloud libraries, and Terraform itself use it).
  - The additional implementation complexity is modest (a decorator or wrapper function) and pays off in dramatically improved robustness.
- **Consequences**:
  - **Positive**: The IaC engine becomes significantly more resilient to transient cloud issues without user intervention.
  - **Positive**: Retry logic is centralized in one place (the Executor's wrapper), ensuring consistent behavior across all providers.
  - **Negative**: Apply operations may take longer to complete when retries occur (but this is preferable to failure).
  - **Negative**: Requires careful configuration of maximum retry attempts and maximum delay to avoid excessively long hangs (e.g., cap total retry time at 5-10 minutes).

Option	Pros	Cons	Chosen?
No retries	Simple; fails fast	Not resilient; frequent user intervention needed	No
Fixed-interval retries	Some resilience; simple	Can cause retry synchronization; not adaptive	No
Exponential backoff with jitter	Highly resilient; avoids retry storms; industry standard	Slightly more complex to implement	**Yes**

Common Pitfalls

⚠️ Pitfall: Rate Limit Ignorance Leads to Global API Ban

- **Description**: Implementing retries without respect to cloud API rate limits (e.g., AWS throttling) can cause the provider to temporarily ban your account's API key if you send too many requests too quickly.
- **Why it's wrong**: The retry logic itself becomes a denial-of-service attack on the cloud API, triggering stricter rate limits or temporary suspensions.
- **Fix**: Always inspect error responses for rate limit headers (e.g., `Retry-After`, `X-RateLimit-Reset`). When a 429 (Too Many Requests) is received, extract the recommended wait time and honor it. Additionally, implement client-side rate limiting (token bucket) for each provider to stay below known limits.

⚠️ Pitfall: Read-After-Create Race with Eventual Consistency

- **Description**: After a `create` call returns successfully, immediately calling `read` to refresh state might return a 404 (Not Found) because the cloud's internal consistency model hasn't propagated the resource existence to all regions or servers.
- **Why it's wrong**: The Executor may incorrectly assume the create failed, or may populate the state file with incomplete data, causing downstream errors.
- **Fix**: In the `create` flow, wrap the post-creation `read` call in its own retry loop with a short delay, specifically handling 404s as transient errors for a limited time (e.g., up to 30 seconds). This is separate from the general retry logic for the `create` itself.

⚠️ Pitfall: Zombie Resources from Partial Apply Failures

- **Description**: If the `apply_plan` fails halfway through (e.g., network partition), some resources may have been created in the cloud but not recorded in the updated state file because the overall apply wasn't committed.
- **Why it's wrong**: On the next plan, the engine sees a discrepancy: the cloud has resources the state file doesn't know about, leading to confusion (should they be imported or deleted?). This is a form of state drift.
- **Fix**: Implement a two-phase approach: 1) Perform all operations, collecting results, but don't persist state yet. 2) If *any* operation failed critically, attempt to roll back created resources (in reverse dependency order) before returning. If rollback succeeds, state remains unchanged. If rollback fails, alert the user about orphaned resources that require manual cleanup. For simplicity in the learning project, we may accept this risk and document it, requiring manual `import` for orphaned resources.

⚠️ Pitfall: Assuming Idempotency Without Validation

- **Description**: Assuming that a cloud API's `delete` operation is idempotent (returns success for a non-existent resource) without verifying it, leading to errors when a provider behaves differently.
- **Why it's wrong**: Not all cloud APIs are perfectly idempotent. Some may return 404 on delete of a non-existent resource, which should be treated as success but might be flagged as an error by naive error detection.
- **Fix**: In the `delete` implementation, catch specific "not found" error codes and treat them as success. Document idempotency expectations as part of the `BaseProvider` contract, and test each provider implementation to ensure they comply.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Provider Interface	Python `abc.ABC` with abstract methods	Protocol classes (structural subtyping) with `typing.Protocol`
HTTP Client for Cloud APIs	`requests` library with session management	`httpx` with async/await support for concurrent operations
Retry Logic	Custom decorator with `time.sleep`	`tenacity` library (rich feature set) or `backoff`
Configuration	JSON/YAML files via `pyyaml`	Dynamic loading with environment variable interpolation
Concurrency	Sequential execution (simpler)	`concurrent.futures.ThreadPoolExecutor` for parallel independent resources

B. Recommended File/Module Structure

```
iac_engine/
├── providers/              # Provider abstraction and implementations
│   ├── __init__.py
│   ├── base.py             # BaseProvider abstract class, ProviderConfig
│   ├── aws/                # AWS-specific provider
│   │   ├── __init__.py
│   │   ├── provider.py     # AWSProvider class
│   │   └── resources/      # Resource-specific modules (ec2, s3, etc.)
│   ├── azure/              # Azure provider (similar structure)
│   └── mock/               # Mock provider for testing
├── executor.py             # Executor class, ApplyResult
├── retry.py                # Retry decorator utility
├── state_manager.py        # (From Milestone 2)
├── planner.py              # (From Milestone 3)
└── cli.py                  # CLI commands (plan_command, apply_command)
```

C. Infrastructure Starter Code

Complete Retry Decorator (`retry.py`):

PYTHON

```python
import time

import random

from typing import Callable, Any, Type, Tuple

from functools import wraps


def retry_with_backoff(

    max_retries: int = 3,

    base_delay: float = 1.0,

    max_delay: float = 30.0,

    jitter: bool = True,

    retry_on_exceptions: Tuple[Type[Exception], ...] = (Exception,)

):

    """

    Decorator that retries a function with exponential backoff and jitter.



    Args:

        max_retries: Maximum number of retry attempts (total calls = max_retries + 1)

        base_delay: Base delay in seconds for exponential backoff (e.g., 1.0 for 1s, 2s, 4s...)

        max_delay: Maximum delay between retries in seconds

        jitter: If True, add random jitter to avoid thundering herd

        retry_on_exceptions: Tuple of exception types to retry on (others will propagate immediately)

    """

    def decorator(func: Callable) -> Callable:

        @wraps(func)

        def wrapper(*args, **kwargs) -> Any:

            retries = 0

            while True:

                try:

                    return func(*args, **kwargs)

                except retry_on_exceptions as e:

                    retries += 1

                    if retries > max_retries:

                        raise  # Re-raise the last exception after exhausting retries



                    # Calculate delay with exponential backoff

                    delay = min(base_delay * (2 ** (retries - 1)), max_delay)



                    # Add jitter (randomize between 0.5*delay and 1.5*delay)

                    if jitter:
```

```
                    delay = delay * (0.5 + random.random())


                # Wait before retrying
                time.sleep(delay)
        return wrapper
    return decorator
```

Mock Provider for Testing (`providers/mock/provider.py`):

PYTHON

```python
from typing import Dict, Optional

from ..base import BaseProvider, Resource


class MockProvider(BaseProvider):

    """Mock provider that stores resources in memory for testing."""



    def __init__(self, config: dict):

        self.config = config

        self.resources: Dict[str, Resource] = {}  # resource_id -> Resource



    def validate_credentials(self, config: dict) -> bool:

        return True  # Mock always validates



    def create(self, resource: Resource) -> Resource:

        # Generate a mock ID

        resource.attributes["id"] = f"mock-{resource.type}-{resource.name}"

        resource_id = resource.attributes["id"]



        # Store in memory

        self.resources[resource_id] = resource

        return resource



    def read(self, resource_id: str, resource_type: str) -> Optional[Resource]:

        return self.resources.get(resource_id)



    def update(self, resource_id: str, resource: Resource) -> Resource:

        if resource_id not in self.resources:

            raise ValueError(f"Resource {resource_id} not found")



        # Update attributes

        current = self.resources[resource_id]

        current.attributes.update(resource.attributes)

        return current



    def delete(self, resource_id: str, resource_type: str) -> bool:

        if resource_id in self.resources:

            del self.resources[resource_id]

            return True

        return True  # Idempotent: deleting non-existent is success
```

D. Core Logic Skeleton Code

BaseProvider Abstract Class (`providers/base.py`):

PYTHON

```python
from abc import ABC, abstractmethod
from typing import Optional, Dict, Any
from dataclasses import dataclass
from ..data_model import Resource  # Assuming Resource is defined elsewhere

@dataclass
class ProviderConfig:
    provider_type: str  # e.g., "aws", "azure"
    config: Dict[str, Any]  # provider-specific configuration

class BaseProvider(ABC):
    """Abstract base class that all cloud providers must implement."""

    @abstractmethod
    def validate_credentials(self, config: dict) -> bool:
        """
        Validate that the provider configuration is correct.

        Returns:
            True if credentials are valid, raises an exception otherwise.
        """
        pass

    @abstractmethod
    def create(self, resource: Resource) -> Resource:
        """
        Create a new resource in the cloud.

        Args:
            resource: Resource object with desired attributes.

        Returns:
            Resource object with actual created attributes (including ID).

        Raises:
            ProviderError: If creation fails permanently.
        """
        pass

    @abstractmethod
```

```
    def read(self, resource_id: str, resource_type: str) -> Optional[Resource]:
        """
        Read the current state of a resource from the cloud.

        Args:
            resource_id: Unique identifier of the resource in the cloud.
            resource_type: Type of the resource (e.g., 'aws_instance').

        Returns:
            Resource object if found, None if the resource does not exist.
        """
        pass

    @abstractmethod
    def update(self, resource_id: str, resource: Resource) -> Resource:
        """
        Update an existing resource in the cloud.

        Args:
            resource_id: Unique identifier of the resource to update.
            resource: Resource object with desired attributes.

        Returns:
            Updated Resource object with actual attributes.
        """
        pass

    @abstractmethod
    def delete(self, resource_id: str, resource_type: str) -> bool:
        """
        Delete a resource from the cloud.

        Args:
            resource_id: Unique identifier of the resource to delete.
            resource_type: Type of the resource.

        Returns:
            True if deletion succeeded or resource was already absent.
            False on unrecoverable failure.
```

```
    """
    pass
```

Executor Class (`executor.py`):

PYTHON

```python
from typing import Dict, List, Optional

from dataclasses import dataclass

from .retry import retry_with_backoff

from .data_model import PlanAction, ActionType, Resource

from .providers.base import BaseProvider


@dataclass

class ApplyResult:

    resource_address: str

    success: bool

    new_state: Optional[Resource]

    error: Optional[str] = None

    retries: int = 0


class Executor:

    """Orchestrates the application of an execution plan using providers."""



    def __init__(self, max_concurrency: int = 1):

        self.max_concurrency = max_concurrency

        self._state_snapshot: Dict[str, Resource] = {}  # In-memory state during apply



    def apply_plan(

        self,

        plan: Dict[str, PlanAction],

        providers: Dict[str, BaseProvider],

        state_before: Dict[str, Resource]

    ) -> Dict[str, ApplyResult]:

        """

        Apply the execution plan to reach desired infrastructure state.



        Args:

            plan: Dictionary mapping resource addresses to PlanAction objects.

                 Assumed to be topologically sorted by the Planner.

            providers: Dictionary mapping provider type strings to BaseProvider instances.

            state_before: The state as it was before apply (from state file).



        Returns:

            Dictionary mapping resource addresses to ApplyResult objects.

        """

        results = {}
```

```python
    # TODO 1: Initialize in-memory state snapshot with state_before
    # TODO 2: Iterate through plan items in order (already sorted by dependencies)
    # TODO 3: For each PlanAction, extract provider type from resource.type
    # TODO 4: Get the appropriate provider from providers dictionary
    # TODO 5: Call _execute_action(action, provider)
    # TODO 6: If action succeeded, update in-memory state snapshot with new_state
    # TODO 7: Store result in results dictionary
    # TODO 8: If any action failed critically, consider stopping early (optional)
    # TODO 9: Return results dictionary
    return results


def _execute_action(
    self,
    action: PlanAction,
    provider: BaseProvider
) -> ApplyResult:
    """
    Execute a single PlanAction with retry logic and error handling.

    Args:
        action: The PlanAction to execute.
        provider: The provider instance to use.

    Returns:
        ApplyResult indicating success/failure.
    """
    resource_address = f"{action.resource.type}.{action.resource.name}"

    @retry_with_backoff(max_retries=3, base_delay=1.0)
    def _call_with_retry():
        # TODO 1: Based on action.action_type, call appropriate provider method
        #   - CREATE: provider.create(action.resource)
        #   - UPDATE: provider.read() then provider.update()
        #   - DELETE: provider.delete()
        #   - NOOP: return existing state (from prior_state or read)
        # TODO 2: For CREATE and UPDATE, call provider.read() after to refresh state
        # TODO 3: Return the resulting Resource (or None for DELETE)
        pass
```

```
        try:
            # TODO 4: Call _call_with_retry() and capture result
            # TODO 5: Return ApplyResult with success=True and new_state
            pass
        except Exception as e:
            # TODO 6: Handle specific exception types:
            #   - Transient errors (network, rate limit): should have been retried
            #   - Permanent errors: capture in ApplyResult with success=False
            # TODO 7: Return ApplyResult with success=False and error message
            pass
```

Sample AWS Provider Skeleton (`providers/aws/provider.py`):

PYTHON

```python
import boto3

from typing import Optional, Dict, Any

from ..base import BaseProvider

from ...data_model import Resource


class AWSProvider(BaseProvider):

    """AWS cloud provider implementation."""



    def __init__(self, config: dict):

        self.config = config

        self.session = boto3.Session(

            aws_access_key_id=config.get("access_key"),

            aws_secret_access_key=config.get("secret_key"),

            region_name=config.get("region", "us-east-1")

        )

        # TODO: Initialize service clients (ec2, s3, etc.) as needed



    def validate_credentials(self, config: dict) -> bool:

        # TODO 1: Attempt a lightweight API call (e.g., sts:GetCallerIdentity)

        # TODO 2: Return True if successful, raise exception with details if not

        pass



    def create(self, resource: Resource) -> Resource:

        resource_type = resource.type

        # TODO 1: Dispatch to appropriate resource-specific create method

        #   Example: if resource_type == "aws_instance": return self._create_ec2_instance(resource)

        # TODO 2: Extract parameters from resource.attributes

        # TODO 3: Call AWS API (e.g., ec2.run_instances)

        # TODO 4: Extract cloud-assigned ID from response

        # TODO 5: Return updated Resource with ID and other attributes

        pass



    def read(self, resource_id: str, resource_type: str) -> Optional[Resource]:

        # TODO 1: Parse resource_type to determine AWS service and method

        # TODO 2: Call appropriate describe/GET API

        # TODO 3: If resource exists, map API response to Resource attributes

        # TODO 4: If resource not found (404/InvalidParameter), return None

        pass
```

```
    # Similar skeletons for update() and delete()
```

E. Language-Specific Hints

- **Abstract Base Classes**: Use Python's `abc` module to define `BaseProvider`. Decorate methods with `@abstractmethod` to enforce implementation in subclasses.
- **Error Handling**: Define a custom exception hierarchy: `ProviderError` as base, with subclasses like `TransientProviderError` (retryable) and `PermanentProviderError`.
- **Type Hints**: Use `typing` module extensively for better IDE support and documentation (e.g., `Optional[Resource]`, `Dict[str, BaseProvider]`).
- **Configuration Management**: Use `pydantic` for validation if advanced configuration schemas are needed, but simple `dict` is acceptable for the learning project.
- **Dependency Injection**: Pass provider instances to the Executor rather than having it instantiate them. This makes testing easier (you can pass mock providers).

F. Milestone Checkpoint

To verify Milestone 4 (Provider Abstraction & Executor) is complete:

1. **Create a test configuration** with a simple resource (e.g., a mock "file" resource that creates a local file).
2. **Run the plan command**: `python cli.py plan config.yaml`. It should show a plan to create the resource.
3. **Apply the plan**: `python cli.py apply config.yaml --auto-approve`. The console should output:

```
Applying plan...
file.example: Creating... (success after 0.2s)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

4. **Verify the resource was created**: Check that the mock resource (e.g., a file on disk) exists with the expected attributes.
5. **Run plan again**: Should show "No changes" because the state matches the configuration.
6. **Test error handling**: Temporarily break the provider's credentials or network connection and verify that appropriate error messages are shown (not stack traces).

Expected Behavior: The Executor calls provider methods in dependency order, respects retry logic, updates the state file upon successful completion, and provides clear feedback about what succeeded or failed.

Interactions and Data Flow

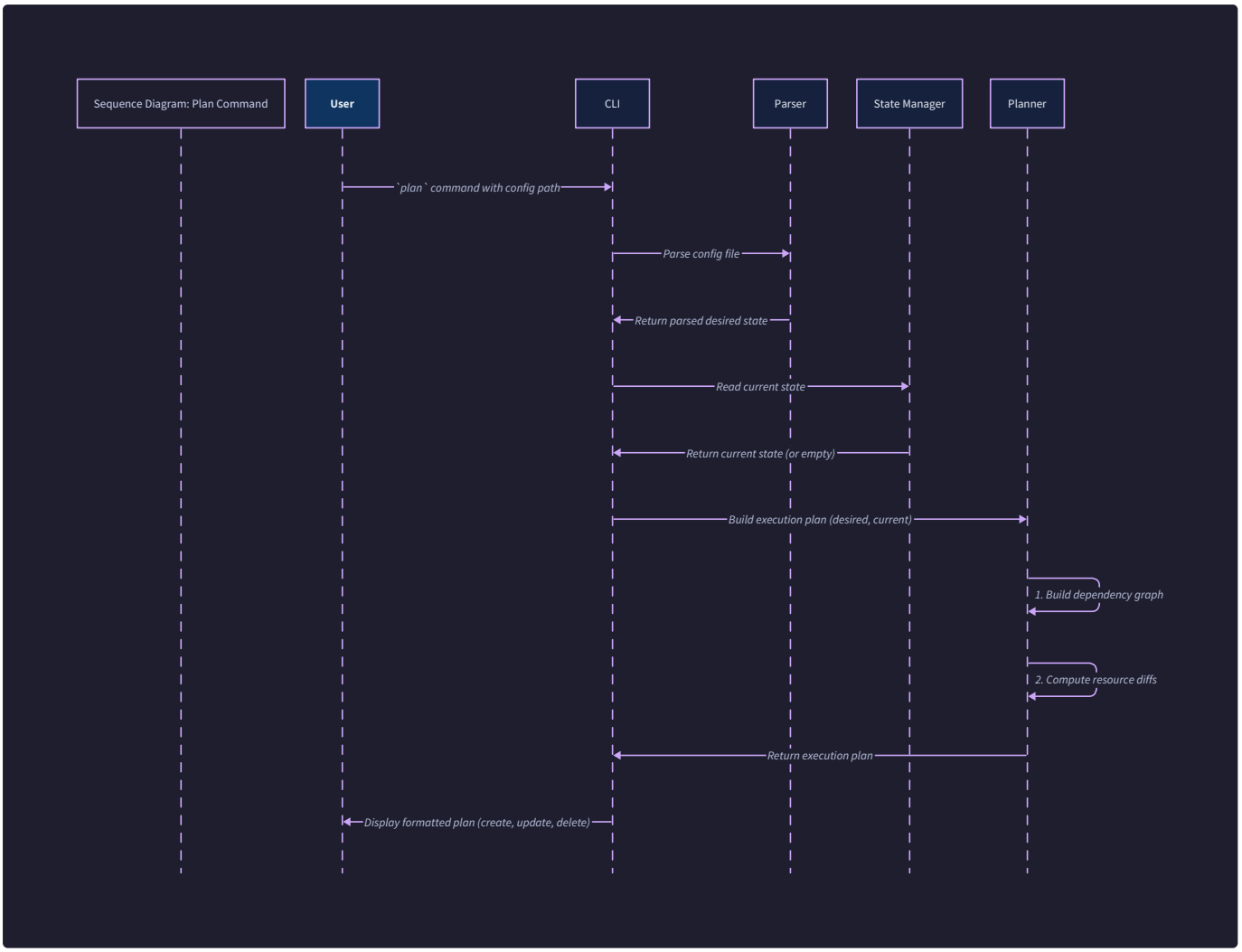
> **Milestone(s):** 3, 4

This section describes the orchestration between components for the two primary workflows in the IaC engine: generating an execution plan (`plan`) and applying that plan to actual infrastructure (`apply`). Understanding these workflows is critical because they represent the main value proposition of the system—transforming declarative configuration into safe, ordered operations on cloud resources.

The mental model for these workflows is that of a **construction project management office**. The `plan` command is the project planning phase: architects review blueprints, check what's already built, and create a detailed work order. The `apply` command is the construction phase: site managers lock the site, hand work orders to specialized crews (providers), and update the building log as work completes.

Sequence: The Planning Workflow

The `plan` workflow answers the question: "What changes would be made if I applied this configuration right now?" It performs a **dry-run analysis** without modifying any infrastructure, which is crucial for safety and review. This workflow combines the **Parser**, **State Manager**, and **Planner** components to produce a human-readable diff of proposed changes.



The workflow begins when a user runs `plan_command(config_path, state_path, var_file)` from the CLI. Here's the step-by-step sequence:

1. **Configuration Loading and Parsing**
   - The CLI calls `process_configuration(root_file, variable_files, cli_vars)`, which orchestrates the Parser component.
   - Internally, `parse_file` reads the root HCL/YAML file and produces a raw abstract syntax tree (AST).
   - `resolve_variables` walks the AST, replacing `${var.name}` references with actual values from variable files and CLI arguments.
   - For each module block found, `load_module` is called recursively to load and resolve the module's resources, flattening them into a single list.
   - The final output is a normalized `List[Resource]` representing the **desired state**—what the infrastructure should look like.
2. **Current State Loading**
   - The CLI calls `read_state(state_path)` on the State Manager.
   - This loads the persistent state file from disk (or remote backend) into a `Dict[str, StateRecord]` dictionary keyed by resource address.
   - If no state file exists, an empty dictionary is returned, representing a "clean slate" with no existing infrastructure.
3. **Dependency Graph Construction**
   - The CLI passes both the desired resources and current state to the Planner via `build_graph(resources, state_records)`.
   - The algorithm extracts dependencies from two sources:
     - **Explicit dependencies**: From `depends_on` attributes in resource configurations.
     - **Implicit dependencies**: By analyzing attribute references (e.g., when Resource B's `vpc_id` references `aws_vpc.main.id`, a dependency edge is created from B → A).
   - The function returns a `Dict[str, DependencyGraphNode]` where each node contains `depends_on` and `required_by` lists.
   - `validate_acyclic(graph)` ensures no circular dependencies exist (which would make topological sorting impossible).
4. **Topological Sorting and Plan Generation**
   - If the graph is acyclic, `topological_sort(graph)` produces a linear ordering where dependencies come before dependents.

- This sorted list is passed to `generate_plan(sorted_resources, desired_resources, current_state)`.
- For each resource address, the function compares the desired `Resource` against the current `StateRecord` (if any) to determine the appropriate `ActionType`:
  - **CREATE**: No state record exists for this resource.
  - **UPDATE**: State record exists but attributes differ (and update is supported).
  - **DELETE**: State record exists but no corresponding desired resource (resource was removed from config).
  - **NOOP**: State matches desired configuration exactly.
- The function assembles these into a `Dict[str, PlanAction]`—the **execution plan**.

5. **Plan Presentation**
   - The CLI formats the plan for human consumption, typically showing:
     - Summary counts of creates, updates, deletes.
     - Detailed diff for each resource showing attribute changes.
     - Visual indication of dependency order.
   - The plan is printed to stdout but **not persisted**—it's ephemeral and tied to the current state snapshot.

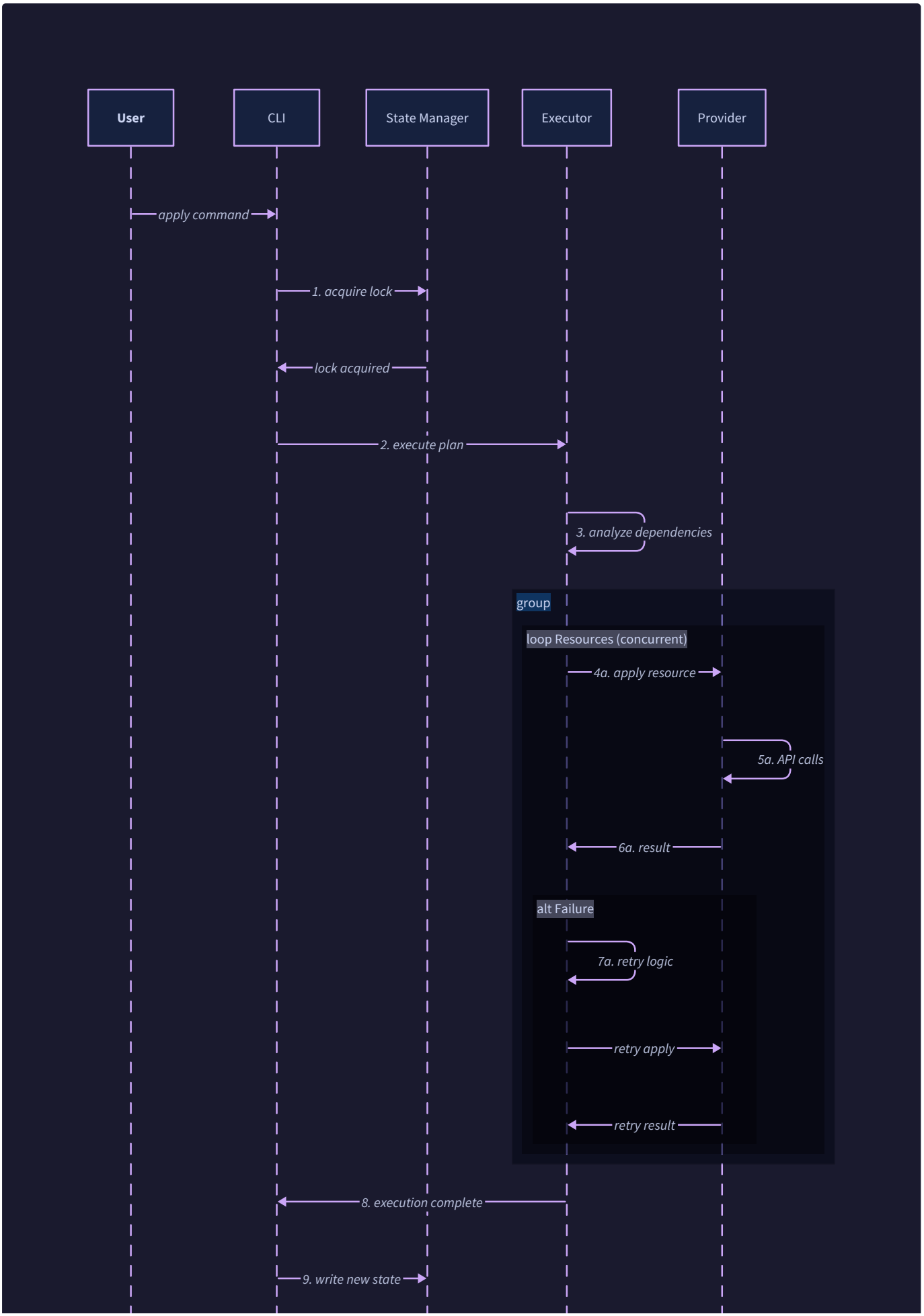
> **Critical Insight:** The plan is a **point-in-time snapshot** based on the state file at the moment of reading. If the state changes between planning and applying (e.g., another engineer applies changes), the plan becomes stale and potentially dangerous to execute. This is why the `apply` workflow includes locking.

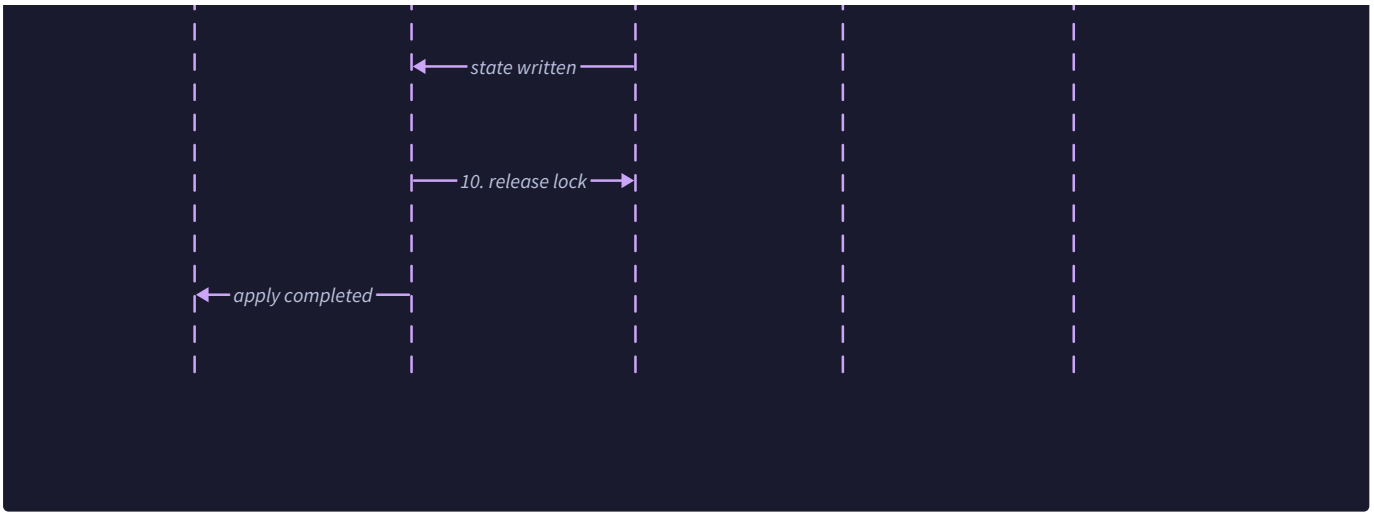
Common Scenario Walkthrough: Consider adding a new web server behind an existing load balancer. The configuration defines an `aws_instance.web` (new) and an `aws_lb_listener.http` (modified to point to the new instance). The planning workflow would:

1. Parse both resources, noting that the listener references the instance's ID via interpolation.
2. Load current state showing the listener exists but the instance doesn't.
3. Build a graph with edge: listener → instance (due to the ID reference).
4. Topologically sort: [instance, listener] (instance must be created first so its ID is available).
5. Generate plan: CREATE for instance, UPDATE for listener.
6. Display: "Plan: 1 to add, 1 to change, 0 to destroy."

Sequence: The Apply Workflow

The `apply` workflow executes the planned changes against real infrastructure in a safe, ordered manner. This is the **moment of truth** where the IaC engine interacts with cloud APIs to create, modify, or destroy resources. It coordinates the **State Manager**, **Executor**, and **Provider** components while maintaining safety through locking and idempotent operations.


User
CLI
State Manager
Executor
Provider
apply command
1. acquire lock
lock acquired
2. execute plan
3. analyze dependencies
group
loop Resources (concurrent)
4a. apply resource
5a. API calls
6a. result
alt Failure
7a. retry logic
retry apply
retry result
8. execution complete
9. write new state



The workflow begins when a user runs `apply_command(config_path, state_path, var_file, auto_approve)`. If `auto_approve` is false, the system first displays the plan and prompts for confirmation. Once approved:

1. **State Lock Acquisition**
   - Before any changes, `acquire_lock(lock_path)` is called to obtain an exclusive lock on the state.
   - This implements **pessimistic concurrency control**: only one apply operation can proceed at a time.
   - The lock includes a heartbeat mechanism to detect and release stale locks from crashed processes.
   - If locking fails (e.g., timeout or another process holds the lock), the apply aborts immediately.
2. **Fresh State Read and Plan Regeneration**
   - With the lock held, `read_state(state_path)` loads the **current state** again.
   - The configuration is reparsed (same as planning phase) to get fresh desired resources.
   - `generate_plan` is called again with the freshly loaded state and desired resources.
   - This **re-planning** ensures the execution plan reflects the absolute latest state, guarding against the "stale plan" problem.
   - If the regenerated plan differs significantly from the originally shown plan (beyond a threshold), some systems ask for reconfirmation.
3. **Plan Execution Orchestration**
   - The plan and provider configurations are passed to `apply_plan(plan, providers, max_concurrency)`.
   - The Executor processes resources in **topological order** (dependencies before dependents), but within independent branches, it can execute concurrently up to `max_concurrency`.
   - For each `PlanAction`, the Executor:
     1. Identifies the appropriate provider based on resource type (e.g., `aws_instance` → AWS provider).
     2. Calls `_execute_action(action, provider)` which delegates to the provider's CRUD methods:
        - **CREATE**: `provider.create(resource)` → returns actual resource with cloud-assigned IDs.
        - **UPDATE**: `provider.update(resource_id, resource)` → returns updated resource.
        - **DELETE**: `provider.delete(resource_id, resource_type)` → returns success/failure.
     3. Implements **retry logic** with exponential backoff for transient failures (rate limits, network issues).
     4. Handles **eventual consistency** by polling `provider.read()` after mutations until desired state is observed.
   - Each action result is captured in an `ApplyResult` tracking success/failure, error messages, and retry counts.
4. **State Persistence and Lock Release**
   - As resources are successfully created/updated, their new attributes are recorded in an **in-memory state dictionary**.
   - For deletions, the corresponding state records are removed.
   - Once all actions complete (or fail), `write_state(state_path, state_data)` atomically writes the updated state to disk.
   - The atomic write (via rename) ensures state file integrity even if the process crashes during write.
   - Finally, `release_lock(lock_handle)` releases the lock, allowing other operations to proceed.
5. **Result Reporting**
   - The CLI summarizes the apply results: counts of successful/errored resources.
   - For failures, it displays error details and may suggest remediation steps.
   - The updated state file now reflects the actual infrastructure, making subsequent plans accurate.

Error Handling During Apply: The Executor must handle partial failures gracefully. The strategy is **"fail-fast within dependency chain, continue across independent chains"**:

- If a resource creation fails, all resources that depend on it cannot be created (marked as failed).
- Resources in independent dependency chains continue processing.
- The state is only updated for **successfully applied resources**—failed resources remain in their previous state.
- This ensures the state file never reflects partially created dependency chains.

Common Scenario Walkthrough: Continuing the web server example, the apply workflow would:

1. Acquire lock on the state file.
2. Re-read state and regenerate plan (same as before if no concurrent changes).
3. Execute in order:
   - **Instance creation**: Call AWS EC2 API, wait for instance to reach `running` state, capture `instance_id`.
   - **Listener update**: Call AWS ELB API to update listener with the new instance's ID.
4. Update state: Add instance record, update listener record with new attributes.
5. Release lock and report: "Apply complete! Resources: 1 added, 1 changed, 0 destroyed."

Key Data Flow Transformations

Throughout these workflows, data transforms through specific stages:

Stage	Input	Processing	Output
Parsing	HCL/YAML files + variables	Lexical analysis, interpolation, module resolution	`List[Resource]` (desired state)
State Loading	State file bytes	JSON deserialization, validation	`Dict[str, StateRecord]` (current state)
Diff Computation	Desired resources + current state	Attribute-by-attribute comparison	`Dict[str, PlanAction]` (change set)
Graph Building	Resources + state	Reference analysis, edge creation	`Dict[str, DependencyGraphNode]` (dependency DAG)
Plan Execution	Plan + providers	CRUD API calls, retry logic	`Dict[str, ApplyResult]` (execution outcomes)
State Update	Current state + execution results	Record addition/modification/removal	Updated `Dict[str, StateRecord]` (new state)

> **Architecture Insight:** The separation between planning and applying is not just organizational—it enables critical safety properties. The planning phase is **pure and deterministic** (no side effects), making it safe to run frequently for review. The applying phase is **transactional** with locking and atomic writes, ensuring consistency despite concurrent access and partial failures.

Common Integration Pitfalls

⚠️ Pitfall: Plan-Apply Race Conditions

- **Description:** User A runs `plan`, then user B applies changes before user A applies. User A's plan is now stale and applying it could overwrite B's changes.
- **Why it's wrong:** The state has changed since the plan was generated, making the plan's assumptions invalid.
- **Fix:** The `apply` workflow always re-reads state and re-plans after acquiring the lock, ensuring it acts on current reality.

⚠️ Pitfall: Locking Granularity Issues

- **Description:** Locking the entire state file for a small change (one resource) blocks all other operations unnecessarily.
- **Why it's wrong:** Reduces system throughput and creates contention.
- **Fix:** In advanced implementations, consider fine-grained locking per resource or dependency subgraph. For our educational context, whole-state locking is acceptable.

⚠️ Pitfall: Missing Rollback on Partial Failure

- **Description:** If the apply fails midway, some resources are created but others aren't, leaving infrastructure in an inconsistent state.
- **Why it's wrong:** The user's configuration doesn't match reality, and dependencies may be broken.

- **Fix:** Implement **reverse execution** for failed chains: if CREATE fails, delete any successfully created dependencies in reverse order. Alternatively, adopt a transactional approach where all changes are staged and committed atomically (complex for cloud APIs).

Implementation Guidance

This section provides concrete implementation patterns for orchestrating the workflows described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Workflow Orchestration	Sequential function calls in CLI	State machine with async/await
Concurrency Control	`threading.Thread` for parallel execution	`asyncio` with semaphores
Retry Logic	Simple loop with `time.sleep()`	`tenacity` library with backoff strategies
Lock Heartbeat	Separate daemon thread	`asyncio` background task

B. Recommended File/Module Structure

```
iac_engine/
├── cli/
│   ├── __init__.py
│   ├── commands.py          # plan_command, apply_command implementations
│   └── formatters.py        # Pretty output for plans and results
├── workflows/
│   ├── __init__.py
│   ├── planner.py           # Orchestrates planning workflow
│   ├── applier.py           # Orchestrates apply workflow
│   └── lock_manager.py      # Lock acquisition/release helpers
├── parser/                  # Milestone 1
├── state/                   # Milestone 2
├── planner/                 # Milestone 3
├── providers/               # Milestone 4
└── executor.py              # Executor component
```

C. Infrastructure Starter Code

Workflow Orchestration Helper:

PYTHON

```python
# workflows/lock_manager.py

import threading

import time

from pathlib import Path

from typing import Optional

from dataclasses import dataclass

import json

import os


@dataclass

class LockHandle:

    lock_path: Path

    process_id: int

    lock_id: str

    heartbeat_thread: Optional[threading.Thread]

    stop_event: threading.Event


class LockManager:

    """Helper for acquiring and releasing locks with heartbeat."""



    @staticmethod

    def acquire_lock(lock_path: Path, timeout_seconds: int = 30,

                     heartbeat_interval: int = 10) -> LockHandle:

        """

        Attempts to acquire an exclusive lock with stale detection.



        Args:

            lock_path: Path to lock file

            timeout_seconds: Maximum time to wait for lock

            heartbeat_interval: How often to update lock timestamp



        Returns:

            LockHandle if acquired, raises TimeoutError otherwise

        """

        start_time = time.time()

        lock_id = f"{os.getpid()}-{int(time.time())}"



        while time.time() - start_time < timeout_seconds:

            try:

                # Try to create lock file atomically
```

```
            with open(lock_path, 'x') as f:
                lock_data = {
                    "process_id": os.getpid(),
                    "lock_id": lock_id,
                    "timestamp": time.time(),
                    "heartbeat_interval": heartbeat_interval
                }
                json.dump(lock_data, f)

            # Start heartbeat thread
            stop_event = threading.Event()
            heartbeat_thread = threading.Thread(
                target=LockManager._heartbeat_worker,
                args=(lock_path, lock_id, heartbeat_interval, stop_event),
                daemon=True
            )
            heartbeat_thread.start()

            return LockHandle(
                lock_path=lock_path,
                process_id=os.getpid(),
                lock_id=lock_id,
                heartbeat_thread=heartbeat_thread,
                stop_event=stop_event
            )

        except FileExistsError:
            # Lock exists, check if stale
            try:
                with open(lock_path, 'r') as f:
                    existing_lock = json.load(f)

                lock_age = time.time() - existing_lock["timestamp"]
                max_age = existing_lock["heartbeat_interval"] * 3

                if lock_age > max_age:
                    # Stale lock, remove it and retry
                    os.remove(lock_path)
                    continue
```

```
            except (json.JSONDecodeError, KeyError, IOError):
                # Corrupt lock file, remove it
                try:
                    os.remove(lock_path)
                except OSError:
                    pass
                continue

        time.sleep(0.5)  # Brief pause before retry

    raise TimeoutError(f"Could not acquire lock within {timeout_seconds} seconds")

@staticmethod
def _heartbeat_worker(lock_path: Path, lock_id: str,
                      interval: int, stop_event: threading.Event):
    """Background thread that updates lock timestamp."""
    while not stop_event.is_set():
        try:
            with open(lock_path, 'r') as f:
                lock_data = json.load(f)

            # Only update if we still own the lock
            if lock_data.get("lock_id") == lock_id:
                lock_data["timestamp"] = time.time()
                with open(lock_path, 'w') as f:
                    json.dump(lock_data, f)
        except (IOError, json.JSONDecodeError):
            pass

        stop_event.wait(interval)

@staticmethod
def release_lock(handle: LockHandle):
    """Releases a previously acquired lock."""
    handle.stop_event.set()
    if handle.heartbeat_thread:
        handle.heartbeat_thread.join(timeout=5)
```

```
        try:
            # Verify we still own the lock before removing
            with open(handle.lock_path, 'r') as f:
                lock_data = json.load(f)

            if lock_data.get("lock_id") == handle.lock_id:
                os.remove(handle.lock_path)
        except (IOError, json.JSONDecodeError):
            # Lock file may already be gone
            pass
```

D. Core Logic Skeleton Code

Plan Command Implementation:

PYTHON

```python
# cli/commands.py

from pathlib import Path

from typing import Optional, Dict, List

from iac_engine.parser import process_configuration

from iac_engine.state import read_state, compute_diff

from iac_engine.planner import build_graph, validate_acyclic, topological_sort, generate_plan

from iac_engine.workflows.planner import generate_plan_summary


def plan_command(config_path: Path, state_path: Path,

                 var_file: Optional[Path] = None) -> None:

    """

    CLI command to generate and show an execution plan.



    Args:

        config_path: Path to root configuration file

        state_path: Path to state file

        var_file: Optional path to variable definitions file

    """

    # TODO 1: Collect variable values from var_file and environment

    #   - Parse var_file if provided (could be JSON, YAML, or .tfvars)

    #   - Also check environment variables with TF_VAR_ prefix

    #   - Combine into a single variables dictionary



    # TODO 2: Parse configuration with variables

    #   - Call process_configuration(root_file, variable_files, cli_vars)

    #   - This returns List[Resource] representing desired state



    # TODO 3: Load current state

    #   - Call read_state(state_path)

    #   - Returns Dict[str, StateRecord] (empty dict if file doesn't exist)



    # TODO 4: Build dependency graph

    #   - Call build_graph(desired_resources, current_state)

    #   - Call validate_acyclic(graph) - raise error if cycles detected



    # TODO 5: Generate execution plan

    #   - Call topological_sort(graph) to get execution order

    #   - Convert desired_resources to dict keyed by address

    #   - Call generate_plan(sorted_order, desired_dict, current_state)
```

```
    # TODO 6: Display plan to user
    #   - Call generate_plan_summary(plan_dict) for human-readable output
    #   - Show counts: + add, ~ change, - destroy
    #   - Show detailed diff for each resource
    #   - Color code output (green for add, yellow for change, red for destroy)

    pass
```

Apply Command Implementation:

PYTHON

```python
# cli/commands.py

from iac_engine.workflows.lock_manager import LockManager, LockHandle

from iac_engine.executor import apply_plan

from iac_engine.state import write_state

from iac_engine.workflows.planner import confirm_plan


def apply_command(config_path: Path, state_path: Path,

                  var_file: Optional[Path] = None,

                  auto_approve: bool = False) -> None:

    """

    CLI command to apply changes to reach desired state.



    Args:

        config_path: Path to root configuration file

        state_path: Path to state file

        var_file: Optional path to variable definitions file

        auto_approve: Skip confirmation prompt

    """

    lock_handle: Optional[LockHandle] = None



    try:

        # TODO 1: Generate plan (same as plan_command)

        #   - Parse config, load state, build graph, generate plan

        #   - This plan is for display/confirmation only



        # TODO 2: Show plan and request confirmation

        #   - Unless auto_approve is True, display plan and ask "Do you want to perform these actions?"

        #   - Use confirm_plan(plan_dict) helper

        #   - If user rejects, exit immediately



        # TODO 3: Acquire state lock

        #   - Determine lock file path (typically state_path + ".lock")

        #   - Call LockManager.acquire_lock(lock_path, timeout_seconds=30)

        #   - Store returned LockHandle



        # TODO 4: Re-plan with fresh state (critical!)

        #   - With lock held, re-read state using read_state(state_path)

        #   - Re-parse configuration (variables might have changed)

        #   - Re-generate plan with fresh state
```

```
        #   - Compare with original plan - if significant differences, warn user

        # TODO 5: Execute plan
        #   - Load provider configurations from parsed config
        #   - Initialize provider instances (AWS, GCP, etc.)
        #   - Call apply_plan(plan, providers, max_concurrency=5)
        #   - This returns Dict[str, ApplyResult] for each resource

        # TODO 6: Update state with successful changes
        #   - Start with copy of original state
        #   - For each successful CREATE/UPDATE: add/update StateRecord
        #   - For each successful DELETE: remove StateRecord
        #   - For failed actions: leave state unchanged (resource stays as-is)
        #   - Call write_state(state_path, updated_state) atomically

        # TODO 7: Release lock
        #   - Call LockManager.release_lock(lock_handle)

        # TODO 8: Report results
        #   - Show summary: X created, Y updated, Z destroyed, W failed
        #   - For failures: show error messages and suggest fixes

    except Exception as e:
        # TODO 9: Cleanup on error
        #   - If lock_handle exists, release it
        #   - Print error message with stack trace in debug mode
        #   - Exit with non-zero code

        pass
```

Executor Core Logic:

PYTHON

```python
# executor.py

import concurrent.futures

from typing import Dict, List, Optional

from dataclasses import dataclass

import time

import random


@dataclass

class ApplyResult:

    resource_address: str

    success: bool

    new_state: Optional[Resource]

    error: Optional[str]

    retries: int


class Executor:

    """Orchestrates plan execution using providers."""


    def __init__(self, max_concurrency: int = 5):

        self.max_concurrency = max_concurrency


    def apply_plan(self, plan: Dict[str, PlanAction],

                   providers: Dict[str, BaseProvider],

                   ) -> Dict[str, ApplyResult]:

        """

        Applies execution plan using providers.


        Args:

            plan: Dictionary of PlanActions keyed by resource address

            providers: Dictionary mapping provider types to provider instances


        Returns:

            Dictionary of ApplyResults keyed by resource address

        """

        results = {}


        # TODO 1: Group actions by dependency level

        #   - Create a dependency graph from the plan

        #   - Group resources that can be executed in parallel (same dependency depth)
```

```
        # TODO 2: Process groups sequentially, resources within group in parallel
        #   - For each dependency level (in topological order):
        #   - Create ThreadPoolExecutor with max_concurrency
        #   - Submit _execute_single_action for each resource in level
        #   - Wait for all in level to complete before proceeding to next

        # TODO 3: Handle resource execution with retries
        #   - Each _execute_single_action should:
        #     1. Identify correct provider from resource type
        #     2. For CREATE: call provider.create() with exponential backoff
        #     3. For UPDATE: call provider.update() with exponential backoff
        #     4. For DELETE: call provider.delete() with exponential backoff
        #     5. Capture result in ApplyResult structure

        # TODO 4: Propagate failures to dependents
        #   - If a resource fails, mark all its dependents as "failed due to dependency"
        #   - Skip execution for those dependents

        # TODO 5: Collect and return results

        return results

    def _execute_single_action(self, action: PlanAction,
                               provider: BaseProvider) -> ApplyResult:
        """Internal method to execute single PlanAction with retry logic."""
        # TODO 1: Implement exponential backoff with jitter
        #   - Base delay: 1 second, multiplier: 2, max delay: 30 seconds
        #   - Add random jitter (±10%) to avoid thundering herd

        # TODO 2: For CREATE actions:
        #   - Call provider.create(action.resource)
        #   - If successful, return ApplyResult with new_state from provider
        #   - If rate limited (429), wait and retry with backoff
        #   - If other error, mark as failed after max retries

        # TODO 3: For UPDATE actions:
        #   - Call provider.update(action.prior_state.resource_id, action.resource)
        #   - Handle eventual consistency: read after update until attributes match
```

```
    # TODO 4: For DELETE actions:
    #   - Call provider.delete(action.prior_state.resource_id, action.resource.type)
    #   - Verify deletion by reading until resource not found

    # TODO 5: Track retry count and include in ApplyResult

    return ApplyResult(
        resource_address=get_resource_address(action.resource),
        success=False,
        new_state=None,
        error="Not implemented",
        retries=0
    )
```

E. Language-Specific Hints

- **Concurrent Execution:** Use `concurrent.futures.ThreadPoolExecutor` for parallel execution within dependency levels. Remember that I/O-bound operations (API calls) benefit from threading in Python despite the GIL.
- **Atomic Writes:** Implement `write_atomic_json` using `tempfile.NamedTemporaryFile` with `delete=False`, writing to the temp file, then `os.replace()` to atomically replace the target file.
- **Retry Logic:** Consider using the `tenacity` library for robust retry patterns: `@retry(stop=stop_after_attempt(5), wait=wait_exponential(multiplier=1, min=1, max=30))`.
- **Provider Loading:** Use Python's `importlib` to dynamically load provider modules based on configuration: `importlib.import_module(f"providers.{provider_type}")`.

F. Milestone Checkpoint

After implementing both workflows, verify with this test scenario:

1. **Setup test configuration:**

```hcl
# test.tf
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "example" {
  vpc_id = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}
```

2. **Run planning workflow:**

```bash
python -m iac_engine.cli plan test.tf --state state.json
```

Expected: Plan shows 2 resources to create, with subnet depending on VPC.

3. **Run apply workflow (with mock provider):**

```bash
python -m iac_engine.cli apply test.tf --state state.json --auto-approve
```

Expected: Resources created in correct order, state file updated with resource IDs.

4. **Verify state:**

```bash
cat state.json
```

Expected: JSON file containing both resources with mock IDs and attributes.

Signs of problems:

- "Cycle detected" error when no cycles exist → Dependency extraction is buggy.
- Resources created in wrong order → Topological sort is incorrect.
- State file empty after apply → Atomic write failed or exceptions occurred.
- Lock never released → Heartbeat thread not stopping properly.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Plan shows no changes when changes exist	State diff logic buggy	Add debug logging to `compute_diff`, compare attribute dictionaries	Ensure diff compares all attributes, not just top-level
Apply hangs forever	Deadlock in dependency chain or stuck provider	Check which resource is currently executing, provider logs	Implement timeouts for provider operations
State file corrupted after crash	Partial write during crash	Check for `.tmp` files near state file	Ensure `write_atomic_json` uses temp file + rename
"Lock already held" when no other process running	Stale lock file from crashed process	Check lock file timestamp and process ID	Implement stale lock detection in `acquire_lock`
Resources created but dependencies broken	Execution order incorrect despite correct plan	Log dependency graph and execution order	Verify topological sort respects all edges
Rate limiting errors even with retries	Retry logic too aggressive	Check retry delay values and jitter	Increase initial delay, add jitter, respect Retry-After headers

Error Handling and Edge Cases

> **Milestone(s):** 2, 3, 4

A robust Infrastructure as Code engine must anticipate and gracefully handle failures that can occur at every stage of its operation—from parsing invalid configuration to dealing with unreliable cloud APIs. Unlike traditional applications where errors often stem from internal logic bugs, an IaC engine operates in a **distributed, eventually consistent, and adversarial environment** where external systems can fail independently, network connections can drop, and users can modify infrastructure outside the IaC workflow. This section provides a systematic framework for categorizing these failures and implementing appropriate recovery strategies, ensuring the system remains predictable and safe even when things go wrong.

Mental Model: The Fault-Tolerant Construction Site

Imagine our construction site from earlier analogies, but now we must account for real-world complications:

- **Bad Blueprints:** The architect's drawings contain contradictory instructions (configuration errors).
- **Missing Materials:** Some building materials are out of stock (provider API failures).
- **Weather Delays:** Sudden rain stops work temporarily (transient network issues).
- **Vandals:** Someone modifies the building overnight without telling the site manager (external drift).
- **Partial Collapse:** Halfway through rebuilding a wall, the scaffolding collapses (partial apply failure).

The site manager (our IaC engine) needs procedures for each scenario: rejecting bad blueprints, waiting for materials, rescheduling delayed work, detecting and reconciling unauthorized changes, and safely recovering from partial failures without making things worse.

Error Classification and Recovery Strategy

Errors in an IaC system fall into distinct categories based on their source, persistence, and impact on system safety. Each category requires a different handling strategy:

Error Category	Detection Point	Common Causes	Recovery Strategy	Safety Impact
Configuration Errors	During parsing (`process_configuration`)	Invalid HCL syntax, undefined variable references, circular dependencies in modules, invalid resource schemas	Fail fast with clear error messages pointing to exact file and line. Never proceed with invalid config.	High - Prevents deploying invalid infrastructure.
State Errors	During state read/write (`read_state`, `write_state`)	Corrupted JSON, partial writes from crashed process, stale lock files, version incompatibility	Automatic fallback to backup file, stale lock detection with forced break, human intervention for schema migration.	Critical - Corrupted state can lead to destructive actions.
Planning Errors	During graph building (`build_graph`) or plan generation (`generate_plan`)	Circular dependencies, unresolvable attribute references, provider schema mismatches	Fail during planning with clear diagnostic output (e.g., "Cycle detected: A → B → A"). Never generate invalid plan.	High - Invalid plans could cause unsafe execution order.
Provider Errors (Transient)	During CRUD operations in `_execute_action`	Network timeouts, rate limiting (429), temporary cloud service unavailability (5xx), eventual consistency delays	Exponential backoff with jitter, circuit breaker pattern, retry with increasing delays up to configured maximum.	Medium - May delay apply but preserves safety.
Provider Errors (Permanent)	During CRUD operations or `validate_credentials`	Invalid credentials, insufficient permissions, unsupported resource configuration, quota exceeded	Fail fast with provider-specific error message. Roll back any in-progress changes in the current operation.	High - Continuing would waste resources and fail anyway.
Partial Apply Failures	During `apply_plan` execution	Resource creation succeeds but dependent resource fails, external modification during apply, provider bug leaves resource in zombie state	Stop further execution (fail-fast), preserve intermediate state, provide clear error report with affected resources.	Critical - Leaves infrastructure in inconsistent state.
Concurrency Conflicts	During `acquire_lock` or state read-modify-write	Another process modifying same state simultaneously, S3 eventual consistency returning stale state	Pessimistic locking with lease timeout, state checksum validation (optimistic concurrency), retry with fresh state read.	Critical - Could cause conflicting modifications.

> **Design Insight:** The safety principle of **"fail safe"** is paramount. When uncertain, the system should err on the side of doing nothing rather than risking destructive action. This is why configuration and planning errors cause immediate failure, while transient errors trigger retries rather than abandonment.

Detailed Recovery Procedures

For each error category, we implement specific recovery logic:

1. **Configuration Error Recovery**
   - **Detection:** Syntax validation in `parse_file`, reference resolution in `resolve_variables`, cycle detection in module loading
   - **Action:** Immediately raise a `ConfigurationError` with:
     - File path and line number (if available)
     - Specific error description
     - Suggested fix when possible
   - **State Preservation:** No state modification occurs
2. **Transient Provider Error Recovery**

```
Algorithm: Exponential Backoff with Jitter
1. Initialize attempt counter = 1, base_delay = 1 second, max_delay = 60 seconds
2. Execute the provider operation (create/read/update/delete)
3. If success: return result
4. If failure is transient (timeout, 429, 5xx):
   a. Calculate delay = min(base_delay * 2^(attempt-1) + random_jitter, max_delay)
   b. Wait for delay
   c. Increment attempt counter
   d. If attempts < max_attempts (default 5): goto step 2
   e. Else: raise PermanentProviderError
5. If failure is permanent: raise immediately
```

PROSE

3. **Partial Apply Failure Recovery**
   - **Detection:** `_execute_action` returns error for a resource in the middle of plan execution
   - **Action:**
     1. Stop executing further actions (fail-fast)
     2. Record current progress in a `partial_failure_state` snapshot
     3. For all successfully applied resources, refresh their state from cloud
     4. Write combined state (original succeeded + refreshed new states) to disk
     5. Return detailed error report listing succeeded and failed resources
   - **User Recovery Path:** User must fix underlying issue, then re-run `apply` (which will compute diff from the partially updated state)

Key Edge Cases and Mitigations

Beyond straightforward error categories, several subtle edge cases can undermine the system's correctness if not properly addressed.

Edge Case 1: Partial Apply Failures with Dependency Chains

Scenario: Creating resource B depends on resource A. A creates successfully, but B fails. The infrastructure is now in an inconsistent state—A exists but B doesn't.

Mitigation Strategy:

- **Fail-Fast Stop:** Stop execution immediately when B fails, preventing creation of C, D, etc.
- **State Refresh & Preservation:** Refresh A's actual state from the cloud (in case cloud defaults differ from config), then write state file containing A (with actual attributes) and B (as desired but uncreated). This creates an **intentional drift** recorded in state.
- **Next Apply Safety:** When user re-runs `apply` after fixing B's issue, the planner will:
  1. Compare desired state (A, B) against current state (A exists, B missing)
  2. Generate `NOOP` for A (already exists with correct attributes)
  3. Generate `CREATE` for B (still needs creation)
  4. Execute only the missing B creation

> **Design Insight:** The state file should always reflect **reality as known to the IaC engine**, not wishful thinking. Recording partial failures honestly enables safe recovery.

Edge Case 2: External Resource Modification (State Drift)

Scenario: A system administrator manually deletes a VM via cloud console that's managed by our IaC engine. The next `plan` should detect this drift and either recreate it or remove it from management.

Mitigation Strategy:

- **Refresh Before Plan:** The `plan_command` should optionally call `refresh_state` for all resources to detect external changes.
- **Drift Detection Algorithm:**
  1. Read desired state from configuration
  2. Read recorded state from state file
  3. For each resource, call provider's `read` method to get actual cloud state
  4. Compare:
     - If actual == recorded: `NOOP`
     - If actual == desired but ≠ recorded: `UPDATE` state only (drift corrected externally)
     - If actual ≠ desired and recorded == desired: `UPDATE` or `RECREATE` (external modification)
     - If actual == null but recorded exists: `CREATE` (external deletion)
     - If actual exists but recorded == null: `IMPORT` suggestion (external creation)
- **User Notification:** Clearly distinguish between "changes to reach desired state" and "drift detection" in plan output.

Edge Case 3: Zombie Resources (Created but Unrecorded)

Scenario: During a `create` operation, the provider API returns success but then crashes before returning the full resource attributes. The state file never gets updated with the new resource's ID, creating a "zombie" resource that exists in cloud but isn't managed.

Mitigation Strategy:

- **Two-Phase Create:** Implement create operations as:
  1. Call provider `create` with desired configuration

2. Immediately call provider `read` with the returned ID to get complete attributes
3. Only if both succeed, record the `read` result in state

- **Orphan Detection:** Periodic cleanup job that:
  1. Lists all resources in cloud for a given tag (e.g., `managed_by=our_iac`)
  2. Cross-references with state file
  3. Flags resources in cloud but not in state for manual review
- **Transaction Boundary:** Treat the entire `_execute_action` for a `CREATE` as atomic—either both create and state update succeed, or we retry/rollback.

Edge Case 4: Schema Evolution and Version Compatibility

Scenario: After upgrading to a new version of a provider, resource attributes that were previously optional become required. Existing state files contain the old schema, causing validation errors.

Mitigation Strategy:

- **State Versioning:** Include a `schema_version` field in `StateRecord` and state file metadata.
- **Migration Hooks:** Provider implementations can define migration functions that upgrade state from v1 → v2 → v3.
- **Backward Compatibility:** When reading old state, apply migration before validation.
- **Forward Safety:** Never automatically downgrade state; require explicit user action.

Edge Case 5: Circular Dependencies with Implicit References

Scenario: Resource A references attribute of Resource B, while Resource B references attribute of Resource A, creating a circular dependency that's not explicitly declared with `depends_on`.

Mitigation Strategy:

- **Static Analysis During Planning:** `build_graph` must detect both explicit (`depends_on`) and implicit (attribute reference) dependencies.
- **Cycle Detection Algorithm:** Use depth-first search with temporary marking to detect cycles in combined dependency graph.
- **Clear Error Messages:** When cycle detected, output the cycle path: "A → B → A" and highlight the problematic attribute references.
- **Require Explicit Break:** Force user to redesign configuration to eliminate cycle (e.g., by using `depends_on` to control order, or restructuring resources).

Edge Case 6: Provider API Eventually Consistency

Scenario: After successfully calling `create`, an immediate `read` returns "not found" because the cloud provider's replication hasn't completed.

Mitigation Strategy:

- **Eventual Consistency Awareness:** After mutating operations, implement progressive polling:
  1. Initial read (may fail with "not found")
  2. Wait 1 second, retry read
  3. Exponential backoff up to reasonable limit (e.g., 30 seconds for global resources)
  4. If still not found after max wait, treat as failure (may need recreate)
- **Resource-Specific Timeouts:** Different resource types have different propagation characteristics (e.g., global DNS vs. regional compute).

Edge Case 7: Concurrent Modification by Multiple Engineers

Scenario: Two engineers run `apply` simultaneously on the same infrastructure, both using shared remote state (e.g., S3).

Mitigation Strategy:

- **Pessimistic Locking:** `acquire_lock` creates a lock file with process ID, timestamp, and owner.
- **Lock Heartbeat:** Background thread periodically updates lock timestamp while operation in progress.
- **Stale Lock Detection:** If lock exists but heartbeat is older than timeout (e.g., 10 minutes), assume previous process crashed and allow breaking lock.
- **State Checksum Validation:** Before writing updated state, verify that the state hasn't changed since we read it (optimistic concurrency as backup).

Edge Case 8: Configuration with Count/For_Each Interpolation

Scenario: `count = length(${var.items})` creates chicken-and-egg problem: need variable value to know how many resources, but variables may reference resource attributes.

Mitigation Strategy:

- **Multi-Pass Resolution:** Implement variable resolution in phases:
  1. Resolve all non-resource-dependent variables
  2. Create initial resource graph with unknown `count`

3. If `count` references unresolved value, error with "cannot compute count before resources exist"
4. Require `count` to use only variables, not resource attributes

- **Alternative:** Support `for_each` with explicitly known collections rather than computed lengths.

ADR: Fail-Fast vs. Continue-on-Error for Partial Apply

Decision: Fail-Fast on Partial Apply

- **Context:** When applying a plan with multiple resources, some may fail while others succeed. We must decide whether to continue applying unrelated resources or stop immediately.
- **Options Considered:**
  1. **Fail-Fast:** Stop execution at first failure, preserve partial state, report error.
  2. **Continue-on-Error:** Skip failed resource, continue with others, aggregate all errors at end.
  3. **Dependency-Aware Continue:** Stop only if dependent resources will fail; continue with independent ones.
- **Decision:** Implement Fail-Fast strategy.
- **Rationale:**
  - **Safety First:** Continuing after unknown failure could compound problems (e.g., creating dependent resources that will fail anyway).
  - **Predictability:** Engineers get immediate feedback rather than discovering multiple failures buried in log.
  - **Simpler Recovery:** With fail-fast, the point of failure is clear and state reflects exactly what succeeded.
  - **Industry Standard:** Terraform and most production IaC tools use fail-fast for apply operations.
- **Consequences:**
  - Engineers must fix errors and re-run apply more frequently.
  - Requires careful dependency ordering to minimize cascading failures.
  - Partial state must be accurately preserved for safe resume.

Option	Pros	Cons	Why Not Chosen
Fail-Fast	Safe, predictable, easier to debug, matches user expectations	May stop early for non-critical failures, requires re-running apply	**Chosen** - Safety outweighs convenience
Continue-on-Error	Maximizes progress in single run, good for independent resources	Can create inconsistent state, harder to debug multiple failures	Risk of creating "zombie" resources with missing dependencies
Dependency-Aware	Intelligent compromise, minimizes unnecessary stops	Complex to implement (need real-time dependency analysis), still risky	Added complexity not justified for educational project

Common Pitfalls

⚠️ Pitfall: Silent State Corruption

- **Description:** Writing state with `json.dump()` directly without atomic rename can result in partially written/corrupted state files if process crashes mid-write.
- **Why Wrong:** Corrupted state file on next read causes parser errors or, worse, incorrect diffs leading to destructive actions.
- **Fix:** Always use `write_atomic_json` which writes to temp file then atomically renames.

⚠️ Pitfall: Infinite Retry Loop

- **Description:** Implementing retry logic without maximum attempts or circuit breaker for permanent failures.
- **Why Wrong:** System hangs forever retrying impossible operations (e.g., invalid credentials), wasting resources.
- **Fix:** Implement exponential backoff with jitter AND maximum attempt limit (e.g., 5 attempts). Add circuit breaker pattern to fail fast after repeated failures.

⚠️ Pitfall: Ignoring Stale Locks

- **Description:** Checking only if lock file exists, not checking its timestamp/heartbeat.
- **Why Wrong:** Crashed process leaves lock file forever, blocking all future operations.
- **Fix:** Implement `acquire_lock` with stale detection (e.g., locks older than 10 minutes can be broken with warning).

⚠️ Pitfall: Not Refreshing State After Partial Apply

- **Description:** After resource A succeeds but B fails, writing state with A's *expected* attributes rather than *actual* attributes from cloud.
- **Why Wrong:** Cloud may set default values different from config; next plan will see incorrect "drift."
- **Fix:** After each successful operation, immediately call `refresh_state` and use the returned actual state.

⚠️ Pitfall: Assuming Immediate Consistency

- **Description:** Calling `read` immediately after `create` and treating "not found" as failure.
- **Why Wrong:** Cloud providers have eventual consistency; resource may exist but not be immediately queryable.
- **Fix:** Implement progressive polling with exponential backoff for read-after-write operations.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Error Classification	Custom exception hierarchy with base `IaCError`	Structured error codes with machine-readable metadata
Retry Logic	Manual retry loops with `time.sleep()`	`tenacity` library with decorators and multiple backoff strategies
Circuit Breaker	Simple counter tracking consecutive failures	`pybreaker` library with half-open state automation
State Validation	JSON Schema validation with `jsonschema`	Protocol Buffers with backward compatibility rules

B. Recommended File Structure

```
iac-engine/
  errors/                    # Error hierarchy and utilities
    __init__.py
    exceptions.py           # Base IaCError and all subclasses
    recovery.py             # Retry, circuit breaker utilities
  state/                    # State management
    lock.py                 # LockHandle and acquisition logic
    file_operations.py      # write_atomic_json, read_json_with_backup
  providers/                # Provider implementations
    base.py                 # BaseProvider with error handling mixins
    retry_decorators.py     # @retry_with_backoff decorator
```

C. Infrastructure Starter Code

Complete Retry Decorator with Exponential Backoff:

PYTHON

```python
# errors/recovery.py

import time

import random

from typing import Callable, Any, Type, Tuple

from functools import wraps


def retry_with_backoff(

    max_attempts: int = 5,

    base_delay: float = 1.0,

    max_delay: float = 60.0,

    jitter: bool = True,

    exceptions: Tuple[Type[Exception], ...] = (Exception,)

):

    """

    Decorator that retries a function with exponential backoff and jitter.



    Args:

        max_attempts: Maximum number of attempts before giving up

        base_delay: Base delay in seconds (will be doubled each retry)

        max_delay: Maximum delay in seconds

        jitter: Whether to add random jitter to delays

        exceptions: Tuple of exception types to catch and retry

    """

    def decorator(func: Callable) -> Callable:

        @wraps(func)

        def wrapper(*args, **kwargs) -> Any:

            attempt = 1

            while attempt <= max_attempts:

                try:

                    return func(*args, **kwargs)

                except exceptions as e:

                    # Check if this is a permanent error (not transient)

                    if hasattr(e, 'permanent') and e.permanent:

                        raise



                    # Permanent errors for certain status codes

                    if hasattr(e, 'status_code'):

                        if e.status_code in [400, 401, 403, 404, 409]:

                            raise  # Don't retry client errors
```

```
                    if attempt == max_attempts:
                        raise  # Max attempts reached

                    # Calculate delay with exponential backoff
                    delay = min(base_delay * (2 ** (attempt - 1)), max_delay)

                    # Add jitter (0.5 to 1.5 multiplier)
                    if jitter:
                        delay = delay * (0.5 + random.random())

                    time.sleep(delay)
                    attempt += 1

            # Should never reach here
            raise RuntimeError("Retry logic exhausted unexpectedly")

        return wrapper
    return decorator

class CircuitBreaker:
    """Simple circuit breaker pattern implementation."""

    def __init__(self, failure_threshold: int = 5, reset_timeout: float = 60.0):
        self.failure_threshold = failure_threshold
        self.reset_timeout = reset_timeout
        self.failure_count = 0
        self.last_failure_time = 0
        self.state = "CLOSED"  # CLOSED, OPEN, HALF_OPEN

    def call(self, func: Callable, *args, **kwargs) -> Any:
        current_time = time.time()

        # Check if circuit breaker is OPEN and reset timeout has passed
        if self.state == "OPEN" and current_time - self.last_failure_time > self.reset_timeout:
            self.state = "HALF_OPEN"

        # Reject calls if circuit is OPEN
        if self.state == "OPEN":
```

```
            raise CircuitOpenError("Circuit breaker is OPEN")


        try:
            result = func(*args, **kwargs)
            # Success - reset circuit if it was HALF_OPEN
            if self.state == "HALF_OPEN":
                self.state = "CLOSED"
                self.failure_count = 0
            return result
        except Exception as e:
            self.failure_count += 1
            self.last_failure_time = current_time


            # Trip circuit if threshold reached
            if self.failure_count >= self.failure_threshold:
                self.state = "OPEN"


            raise
```

D. Core Logic Skeleton Code

Error-Aware Execute Action:

PYTHON

```python
# providers/base.py

from errors.recovery import retry_with_backoff, CircuitBreaker

from typing import Optional, Dict, Any


class ProviderError(Exception):

    """Base exception for all provider errors."""

    def __init__(self, message: str, resource_address: str = "", permanent: bool = False):

        super().__init__(message)

        self.resource_address = resource_address

        self.permanent = permanent


class TransientProviderError(ProviderError):

    """Transient error that may succeed on retry."""

    def __init__(self, message: str, resource_address: str = ""):

        super().__init__(message, resource_address, permanent=False)


class PermanentProviderError(ProviderError):

    """Permanent error that won't succeed with retry."""

    def __init__(self, message: str, resource_address: str = ""):

        super().__init__(message, resource_address, permanent=True)


class BaseProvider:

    # ... other methods ...



    @retry_with_backoff(max_attempts=3, base_delay=1.0, max_delay=10.0)

    def create(self, resource: Resource) -> Resource:

        """

        Create a resource in the cloud with retry logic.



        TODO 1: Validate resource configuration matches provider schema

        TODO 2: Call cloud API to create resource (may raise TransientProviderError)

        TODO 3: If API returns success but incomplete data, immediately call self.read()

        TODO 4: Return Resource with actual cloud attributes (not just desired)

        TODO 5: Handle specific error cases:

               - Rate limiting (429) -> raise TransientProviderError

               - Invalid config (400) -> raise PermanentProviderError

               - Timeout -> raise TransientProviderError

        TODO 6: Ensure operation is idempotent (same call twice creates only one resource)

        """

        pass
```

```
    def read(self, resource_id: str, resource_type: str) -> Optional[Resource]:
        """
        Read current resource state from cloud with eventual consistency handling.

        TODO 1: Call cloud API to get resource
        TODO 2: If resource not found, return None
        TODO 3: If API returns 404 but resource was just created, implement progressive polling:
               - Wait 1 second, retry
               - Exponential backoff up to 30 seconds
               - If still not found, assume creation failed
        TODO 4: Convert cloud API response to Resource object
        TODO 5: Handle rate limiting and timeouts with retry decorator
        """
        pass
```

Partial Apply Recovery in Executor:

PYTHON

```python
# executor/apply.py

from typing import Dict, List

from dataclasses import dataclass

from errors.exceptions import PartialApplyError


@dataclass

class ApplyResult:

    resource_address: str

    success: bool

    new_state: Optional[Resource]

    error: Optional[str]

    retries: int


def apply_plan(plan: Dict[str, PlanAction], providers: Dict[str, BaseProvider],

               max_concurrency: int = 10) -> Dict[str, ApplyResult]:

    """

    Apply execution plan with partial failure recovery.



    TODO 1: Group actions by provider for efficient API usage

    TODO 2: For each action in topological order (respecting dependencies):

            - Execute with _execute_action helper

            - If success: refresh state from cloud, store in results

            - If failure:

                a. Stop execution (fail-fast)

                b. Refresh state for all succeeded resources

                c. Compile partial state (original + succeeded refreshed)

                d. Write partial state to disk

                e. Raise PartialApplyError with details

    TODO 3: If all actions succeed: write complete new state

    TODO 4: Return ApplyResult for each resource

    """

    results = {}

    succeeded_resources = {}



    # TODO: Sort actions by dependency order from plan

    sorted_actions = topological_sort_actions(plan)



    for resource_address, action in sorted_actions:

        try:

            result = _execute_action(action, providers)
```

```
            results[resource_address] = result

            if result.success:
                # Refresh to get actual cloud state (not just expected)
                refreshed = refresh_state(result.new_state, providers)
                succeeded_resources[resource_address] = refreshed
            else:
                # Partial failure - stop execution
                raise PartialApplyError(
                    message=f"Resource {resource_address} failed: {result.error}",
                    succeeded=succeeded_resources,
                    failed_resource=resource_address
                )

        except PartialApplyError:
            # Re-raise to outer handler
            raise
        except Exception as e:
            # Unexpected error - convert to PartialApplyError
            raise PartialApplyError(
                message=f"Unexpected error applying {resource_address}: {str(e)}",
                succeeded=succeeded_resources,
                failed_resource=resource_address
            )

    return results
```

E. Language-Specific Hints

- **Python Exception Chaining:** Use `raise NewError("message") from original_error` to preserve stack trace.
- **Context Managers for Cleanup:** Use `with` statements for lock acquisition and file operations to ensure cleanup even on error.
- **Typing for Error Metadata:** Use `typing.TypedDict` or dataclasses for structured error return values.
- **Logging Structured Errors:** Use `logging.exception()` with extra context dict for machine-parsable error logs.

F. Milestone Checkpoint

After implementing error handling for Milestone 4:

1. **Test Configuration Errors:** Run `plan_command` with invalid HCL syntax - should fail immediately with clear error message.
2. **Test Transient Retry:** Mock a provider that fails twice then succeeds - verify retry logic works.
3. **Test Partial Apply:** Create a plan where second resource fails - verify:
   - Execution stops after first failure
   - State file contains only first resource (refreshed from cloud)
   - Error message clearly indicates which resource failed
4. **Test Lock Recovery:** Start an apply, kill process, verify lock is stale-detected after timeout.

Expected Behavior:

```
$ python -m iac_engine plan invalid_config.hcl
ERROR: Configuration error in invalid_config.hcl:10
  Undefined variable "vpc_id" referenced in resource "aws_instance.web"
  Hint: Define variable in variables.tf or pass with -var flag
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Plan shows no changes	State file out of sync with actual cloud	Run `refresh_state` on all resources, compare with state file	Implement state refresh before planning
Lock never released	Process crashed without cleanup	Check lock file timestamp, if >10 min old	Implement stale lock detection in `acquire_lock`
Apply hangs forever	Infinite retry loop or waiting for consistency	Check logs for repeated "retrying" messages	Add max attempt limit, progressive polling timeout
State file corrupted	Process killed during write	Check for `.tmp` files in state directory	Use `write_atomic_json` with temp file + rename
Cycle detected error	Circular dependency in config	Run `build_graph` with debug output	Redesign config to break cycle, use explicit `depends_on`
Resource disappears after create	Eventual consistency not handled	Check timestamps: create succeeded but immediate read failed	Implement read-after-write polling in `create` method

Testing Strategy

> **Milestone(s):** 1, 2, 3, 4

A robust testing strategy is essential for an IaC engine due to the high stakes of infrastructure changes—a bug could cause service outages, security vulnerabilities, or costly cloud resource waste. This section outlines a comprehensive verification approach combining automated tests at multiple levels with manual verification checkpoints for each milestone. The goal is to build confidence that each component behaves correctly in isolation and that the integrated system safely orchestrates infrastructure changes.

Testing Approaches and Tools

The testing strategy follows the **testing pyramid**, emphasizing many fast, isolated unit tests at the base, fewer integration tests in the middle, and a minimal number of slow, realistic end-to-end tests at the top. This balances thoroughness with execution speed, enabling rapid development cycles while catching complex interaction bugs.

Unit Testing

Unit tests verify the behavior of individual functions and classes in isolation, mocking all external dependencies. They should be fast, deterministic, and cover edge cases.

Component	What to Unit Test	Mocking Strategy	Tools/Patterns
Parser	Variable interpolation logic, module resolution, syntax validation	Mock file system I/O for config loading	`unittest.mock` for patching `open`, `os.path`; parameterized tests for different HCL patterns
State Manager	Atomic file write logic, lock acquisition/release, diff computation	Mock file operations and system calls for locks	`tempfile` for isolated test directories; mock `fcntl` or `msvcrt` for platform-specific locking
Planner	Dependency graph construction, cycle detection, topological sort	Provide synthetic resource lists with known dependencies	Graph algorithm verification with known DAGs and cyclic graphs; property-based testing for sort invariants
Provider/Executor	Retry logic, circuit breaker state transitions, action execution flow	Mock cloud API calls with controlled responses	`unittest.mock` for provider methods; simulate API rate limits and transient failures

For unit testing, Python's built-in `unittest` framework combined with `pytest` for richer fixtures and parameterization is recommended. Use `hypothesis` for property-based testing of complex algorithms like topological sort, where you can verify that for any arbitrary DAG, the sort order respects all dependencies.

Integration Testing

Integration tests verify that components work correctly together, with real implementations of some internal dependencies but external systems (like cloud APIs) still mocked.

Integration Scope	Test Focus	Setup Required	Validation
Parser → Planner	That parsed resources correctly feed into graph building	Parse real config files from test fixtures	Verify that implicit dependencies are extracted from attribute references
State Manager → Planner	That state diffs accurately reflect changes between configurations	Create state files, modify configs, compute plan	Plan actions should match expected CREATE/UPDATE/DELETE
Planner → Executor	That execution plan is correctly transformed into provider calls	Mock providers, feed plan to executor	Verify that provider methods are called in correct order with correct arguments
Full Pipeline (no cloud)	End-to-end flow from config to plan without actual provisioning	All components with mocked providers	Plan output should be predictable and safe (no unintended deletions)

A key integration testing pattern is the **"fake provider"**—a complete implementation of the `BaseProvider` interface that simulates a cloud API in memory. This allows testing the entire engine workflow without network calls or cloud costs. The fake provider maintains an in-memory resource store and can be programmed to simulate specific failures (timeouts, rate limits) to test error handling.

End-to-End Testing

End-to-end (E2E) tests run the actual engine against real cloud providers in a controlled, isolated environment (like a dedicated test cloud account). These tests are slow, expensive, and flaky due to cloud eventual consistency, so they should be minimal and focused on critical happy paths and failure recovery scenarios.

E2E Scenario	Infrastructure Goal	Success Criteria	Cleanup
Basic resource lifecycle	Create, update, delete a simple resource (e.g., S3 bucket)	Resource exists with correct attributes after apply; gone after destroy	Engine must destroy all test resources even if test fails
Dependency ordering	Create resources with dependencies (e.g., VPC before subnet)	Resources created in correct order; deletion reverses order	Verify deletion order via provider logs
Failure and recovery	Simulate partial apply failure (e.g., rate limit mid-operation)	Engine stops gracefully; state is not corrupted; retry works	Manual intervention may be needed for zombie resources

For E2E tests, use cloud provider sandbox accounts with tight budget controls and automatic cleanup via tagged resources. Run these tests only in the main branch or release candidates, not on every pull request.

Mocking Cloud APIs

Comprehensive mocking is essential for reliable, fast tests. Two complementary approaches are recommended:

1. Ad-hoc mocking with `unittest.mock`: Suitable for unit and integration tests where you control the exact sequence of calls. You can mock individual methods of a provider to return specific responses or raise exceptions.

2. Structured fake provider: A full in-memory implementation that mimics cloud API behavior, including:

- Resource storage with unique identifiers
- Basic validation (required attributes)
- Simulated eventual consistency (delayed visibility of updates)
- Configurable failure modes (error rates, latency)

The fake provider can be shared across tests and even used as a teaching tool to understand how the engine interacts with providers.

Decision: Fake Provider over HTTP API Mock

- **Context**: We need to test provider interactions without network calls. While HTTP mocks (like `responses` or `httpretty`) can intercept requests, they tie tests to specific HTTP details.
- **Options Considered**:
  1. **HTTP-level mocking**: Intercept HTTP requests to cloud APIs.
  2. **Fake provider**: Implement the `BaseProvider` interface with in-memory storage.
- **Decision**: Implement a fake provider.
- **Rationale**: The fake provider operates at the same abstraction level as real providers (the `BaseProvider` interface), making tests more resilient to changes in HTTP libraries or API endpoints. It's also faster (no network stack) and can simulate higher-level behaviors like eventual consistency more naturally.
- **Consequences**: Tests are more portable and focused on business logic, but the fake provider must be maintained alongside real providers. It may not catch protocol-specific bugs (e.g., authentication headers).

Option	Pros	Cons	Chosen?
HTTP-level mocking (e.g., `responses`)	Catches HTTP-level bugs; works with any HTTP client	Tied to specific HTTP library; verbose to set up; misses interface logic	No
Fake provider (in-memory)	Matches abstraction level; fast; can simulate complex behaviors	Requires maintenance; may drift from real provider behavior	Yes

Milestone Checkpoints and Verification

Each milestone has specific deliverables that should be verified through both automated tests and manual commands. Below are checkpoints for each milestone, including what to test, how to test it, and commands to run for verification.

Milestone 1: Configuration Parser

The parser must correctly interpret configuration files, resolve variables, and expand modules into a flat list of resources.

Automated Test Coverage:

Functionality	Test Cases	Expected Outcome
Basic parsing	Simple HCL/YAML with one resource	Returns list with one `Resource` with correct `type`, `name`, `attributes`
Variable interpolation	Config with `${var.name}` references	Variables replaced with actual values from var files or CLI
Module resolution	Config with `module "x"` block	Module resources are loaded and merged into main resource list
Error handling	Invalid syntax, missing variables, circular module references	Raises `ConfigurationError` with clear message

Manual Verification Checklist:

1. **Create a test configuration** (`test.tf`):

```hcl
variable "instance_type" {
  default = "t2.micro"
}

resource "aws_instance" "web" {
  instance_type = var.instance_type
  ami           = "ami-123456"
}
```

2. **Run the parser from a Python shell or temporary script:**

```python
from parser import process_configuration

resources = process_configuration(Path("test.tf"), [], {})

print(f"Found {len(resources)} resources")

for r in resources:

    print(f"  {r.type}.{r.name}: {r.attributes}")
```

3. **Verify output:**
   - Should print exactly 1 resource: `aws_instance.web`
   - Attribute `instance_type` should be `"t2.micro"` (resolved variable)
   - Attribute `ami` should be `"ami-123456"`
4. **Test variable precedence:** Pass a different `instance_type` via CLI variables dict and confirm it overrides the default.
5. **Test module loading:** Create a module directory and reference it; verify its resources appear in the output.

Command to run unit tests:

```bash
pytest tests/unit/test_parser.py -v
```

Expected: All tests pass, with >80% line coverage for parser module.

Milestone 2: State Manager

The state manager must reliably read/write state files, handle locking, and compute accurate diffs.

Automated Test Coverage:

Functionality	Test Cases	Expected Outcome
Atomic writes	Simulate crash during write (e.g., kill process)	State file remains intact; backup is used for recovery
Locking	Concurrent lock attempts from different processes	Only one acquires lock; others timeout or wait
Diff computation	Various desired vs. current state scenarios	Correct `PlanAction` types (CREATE, UPDATE, DELETE, NOOP)
Remote backend	Mock S3/GCS operations	State can be read from/written to remote location

Manual Verification Checklist:

1. **Create an initial state file** (`state.json`) with one resource:

```json
{
  "aws_instance.web": {
    "resource_id": "i-123456",
    "resource_type": "aws_instance",
    "resource_name": "web",
    "attributes": {"instance_type": "t2.micro", "ami": "ami-123456"},
    "dependencies": []
  }
}
```

2. **Test reading and writing:**

```python
from state_manager import read_state, write_state

state = read_state(Path("state.json"))

print(state["aws_instance.web"]["attributes"]["instance_type"])

# Modify

state["aws_instance.web"]["attributes"]["instance_type"] = "t2.small"

write_state(Path("state.json"), state)

# Verify file updated atomically (check .bak file exists)
```

3. **Test locking:** In two separate terminal sessions, attempt to acquire a lock on the same file. The first should succeed; the second should timeout after 30 seconds.
4. **Test diff:** Create a desired resource list that changes the `instance_type` and adds a new resource. Call `compute_diff` and verify it returns an UPDATE for the instance and a CREATE for the new resource.

Command to run unit tests:

```bash
pytest tests/unit/test_state_manager.py -v
```

Expected: Tests pass, including a test that verifies atomic write by simulating a power failure (writing a large state file and interrupting the process).

Milestone 3: Planner (Dependency Graph & Planning)

The planner must construct a correct DAG, detect cycles, and generate a safe execution plan.

Automated Test Coverage:

Functionality	Test Cases	Expected Outcome
Graph construction	Resources with explicit `depends_on` and implicit references	Graph edges created in both directions
Cycle detection	Resources with circular dependencies	`validate_acyclic` returns `False` or raises exception
Topological sort	Valid DAG with known dependency order	Sort order respects all dependencies
Plan generation	Various state diffs	Plan actions are in correct order (CREATE/UPDATE before DELETE when replacing)

Manual Verification Checklist:

1. **Create a simple dependency scenario:** Two resources where a subnet depends on a VPC (reference in attributes or explicit `depends_on`).
2. **Build graph and validate:**

```python
from planner import build_graph, validate_acyclic, topological_sort

graph = build_graph(resources, current_state)

print(f"Graph has {len(graph)} nodes")

if validate_acyclic(graph):

    order = topological_sort(graph)

    print(f"Execution order: {order}")

else:

    print("Cycle detected!")
```

3. **Verify execution order:** The VPC should appear before the subnet in the sorted list.
4. **Generate a plan:** Use the diff from Milestone 2 and the sorted order to generate a plan. Print the plan and verify actions are in the correct order.
5. **Test cycle detection:** Intentionally create a circular dependency (A depends on B, B depends on A). The validator should catch it.

Command to run unit tests:

```bash
pytest tests/unit/test_planner.py -v
```

Expected: All tests pass, including a property-based test that for any random DAG, the topological sort produces a valid linear ordering.

Milestone 4: Provider Abstraction & Executor

The provider interface must be implemented correctly, and the executor must apply plans with retries and error handling.

Automated Test Coverage:

Functionality	Test Cases	Expected Outcome
Provider CRUD	Mock API calls for each operation	Methods return appropriate `Resource` or success indicator
Retry logic	Simulate transient failures (timeouts, 5xx errors)	Operation retries with exponential backoff, eventually succeeds
Circuit breaker	Repeated provider failures	Circuit opens after threshold, fails fast, resets after timeout
Executor flow	Plan with multiple actions, some failing	Executor applies successful actions, updates state, reports partial failure

Manual Verification Checklist:

1. **Implement a mock provider:** Create a simple in-memory provider (e.g., `MockProvider`) that stores resources in a dict. Test each CRUD method.
2. **Test retry decorator:** Apply `@retry_with_backoff` to a function that fails 3 times then succeeds. Verify it retries exactly 3 times with increasing delays.
3. **Test circuit breaker:** Create a function that always fails. Use `CircuitBreaker` to call it; after N failures, subsequent calls should immediately raise `CircuitOpenError`.
4. **End-to-end integration test:**
    - Parse a config.
    - Read empty state (no resources).
    - Generate a plan (should be all CREATEs).
    - Execute the plan using the mock provider.
    - Verify state file now contains the resources with correct IDs from the mock provider.

Command to run unit tests:

```bash
pytest tests/unit/test_provider.py tests/unit/test_executor.py -v
```

Expected: All tests pass, including an integration test that runs a full `apply_command` with a mock provider and verifies the final state.

Integrated System Verification

After completing all milestones, run the full test suite and perform a manual smoke test:

```bash
# Run all unit and integration tests
pytest tests/ -v --cov=iac_engine --cov-report=html

# Expected: >90% total coverage, all tests pass

# Smoke test: plan and apply a simple configuration
python -m iac_engine plan test_configs/simple.tf
python -m iac_engine apply test_configs/simple.tf --auto-approve
```

The smoke test should show a plan with the expected changes and apply them successfully, updating the state file.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Testing Framework	`pytest` + `unittest.mock`	`pytest` with `pytest-asyncio` for async tests
HTTP Mocking	`requests-mock` for `requests`-based providers	`aioresponses` for async HTTP clients
Property-based Testing	`hypothesis` for generating test cases	Custom generators for complex resource graphs
Coverage Reporting	`pytest-cov` for coverage reports	Integrate with Codecov or Coveralls for CI
End-to-End Testing	Manual scripts with cleanup tags	Automated using Terraform's testing framework as inspiration

B. Recommended File/Module Structure

```
iac_engine/
  src/
    iac_engine/
      parser/           # Milestone 1
        __init__.py
        parser.py
        resolver.py
        module_loader.py
      state/            # Milestone 2
        __init__.py
        manager.py
        lock.py
        backends.py
      planner/          # Milestone 3
        __init__.py
        graph.py
        plan.py
      providers/        # Milestone 4
        __init__.py
        base.py
        executor.py
        retry.py
        circuit_breaker.py
        aws/            # Example provider implementation
          __init__.py
          ec2.py
        mock/           # Fake provider for testing
          __init__.py
          provider.py
      cli.py            # CLI entry point
      exceptions.py     # IaCError and subclasses
  tests/
    unit/
      test_parser.py
      test_state_manager.py
      test_planner.py
      test_providers.py
      test_executor.py
    integration/
      test_parser_integration.py
      test_plan_generation.py
      test_full_workflow.py
    fixtures/           # Sample configs, state files
      configs/
        simple.tf
        with_modules/
      state/
        empty.json
        simple.json
    conftest.py         # Shared pytest fixtures
```

C. Infrastructure Starter Code

Fake Provider for Testing (Complete Implementation):

PYTHON

```python
# tests/fake_provider.py

from typing import Dict, Any, Optional

from iac_engine.providers.base import BaseProvider, Resource


class FakeProvider(BaseProvider):

    """In-memory fake provider for testing."""



    def __init__(self, provider_type: str):

        self.provider_type = provider_type

        self.resources: Dict[str, Resource] = {}  # resource_id -> Resource

        self._next_id = 1



    def validate_credentials(self, config: Dict[str, Any]) -> bool:

        # Always valid for testing

        return True



    def create(self, resource: Resource) -> Resource:

        # Simulate creation with a generated ID

        resource_id = f"{self.provider_type}-{self._next_id}"

        self._next_id += 1



        # Store the resource with its ID

        resource.attributes["id"] = resource_id

        self.resources[resource_id] = resource



        # Simulate eventual consistency by not immediately returning

        # (in real tests, you might add a delay flag)

        return resource



    def read(self, resource_id: str, resource_type: str) -> Optional[Resource]:

        return self.resources.get(resource_id)



    def update(self, resource_id: str, resource: Resource) -> Resource:

        if resource_id not in self.resources:

            raise KeyError(f"Resource {resource_id} not found")



        # Update attributes

        self.resources[resource_id].attributes.update(resource.attributes)

        return self.resources[resource_id]
```

```
    def delete(self, resource_id: str, resource_type: str) -> bool:
        if resource_id in self.resources:
            del self.resources[resource_id]
            return True
        return False
```

Retry Decorator (Complete Implementation):

PYTHON

```python
# src/iac_engine/providers/retry.py

import time

import random

from functools import wraps

from typing import Callable, Type, Tuple, Any


def retry_with_backoff(

    max_attempts: int = 3,

    base_delay: float = 1.0,

    max_delay: float = 10.0,

    jitter: bool = True,

    exceptions: Tuple[Type[Exception], ...] = (Exception,)

):

    """

    Decorator that retries a function with exponential backoff.



    Args:

        max_attempts: Maximum number of attempts (including first)

        base_delay: Base delay in seconds for exponential backoff

        max_delay: Maximum delay in seconds

        jitter: If True, add random jitter to delays

        exceptions: Tuple of exception types to catch and retry on

    """

    def decorator(func: Callable) -> Callable:

        @wraps(func)

        def wrapper(*args, **kwargs) -> Any:

            attempts = 0

            while True:

                try:

                    attempts += 1

                    return func(*args, **kwargs)

                except exceptions as e:

                    if attempts >= max_attempts:

                        raise



                    # Calculate delay with exponential backoff

                    delay = min(base_delay * (2 ** (attempts - 1)), max_delay)



                    # Add jitter (up to 25% of delay)
```

```
                if jitter:
                    delay = delay * (0.75 + 0.25 * random.random())


                time.sleep(delay)
            return wrapper
        return decorator
```

D. Core Logic Skeleton Code

Test for Dependency Graph Construction:

```python
# tests/unit/test_planner.py

import pytest

from iac_engine.planner.graph import build_graph, validate_acyclic

from iac_engine.parser import Resource


def test_build_graph_with_implicit_dependencies():

    """Test that implicit dependencies are extracted from attribute references."""

    # TODO 1: Create two resources where resource B references resource A

    #   resource_a = Resource(type="aws_vpc", name="main", attributes={"id": "vpc-123"})

    #   resource_b = Resource(type="aws_subnet", name="web", attributes={"vpc_id": "${aws_vpc.main.id}"})


    # TODO 2: Build graph with these resources and empty state


    # TODO 3: Verify graph has two nodes


    # TODO 4: Verify edge exists from aws_subnet.web to aws_vpc.main

    #   (subnet depends on vpc)


    # TODO 5: Validate graph is acyclic (should return True)


    pass

def test_cycle_detection():

    """Test that circular dependencies are detected."""

    # TODO 1: Create three resources with circular dependency:

    #   A -> B (A depends on B)

    #   B -> C (B depends on C)

    #   C -> A (C depends on A)


    # TODO 2: Build graph


    # TODO 3: Call validate_acyclic - should return False or raise exception


    # TODO 4: Verify appropriate error message indicates cycle


    pass
```

Test for State Diff Computation:

```python
# tests/unit/test_state_manager.py

def test_compute_diff_scenarios():

    """Test various diff scenarios: create, update, delete, noop."""

    # TODO 1: Setup: Create a current_state dict with one resource


    # TODO 2: Scenario CREATE: desired_resources has a new resource not in current_state

    #   Call compute_diff

    #   Verify result contains PlanAction with action_type=CREATE for that resource


    # TODO 3: Scenario UPDATE: desired_resources modifies an attribute of existing resource

    #   Verify action_type=UPDATE


    # TODO 4: Scenario DELETE: desired_resources removes a resource present in current_state

    #   Verify action_type=DELETE


    # TODO 5: Scenario NOOP: resource identical in current and desired

    #   Verify action_type=NOOP


    # TODO 6: Edge case: resource with dependencies being deleted

    #   Verify delete action includes proper dependency ordering consideration


    pass
```

E. Language-Specific Hints

- Use `pytest.fixture` for shared test setup (e.g., temporary directories, mock providers).
- Use `@pytest.mark.parametrize` to test multiple input/output combinations for parsing functions.
- For testing file operations, use `tempfile.TemporaryDirectory()` to ensure cleanup.
- Mock time in tests involving retries or circuit breakers using `unittest.mock.patch('time.sleep')` and `unittest.mock.patch('time.time')`.
- Use `pytest.raises(Exception)` context manager to verify that expected exceptions are raised.

F. Milestone Checkpoint Commands

After implementing each milestone, run these commands to verify basic functionality:

Milestone 1:

```bash
# Run parser tests

pytest tests/unit/test_parser.py -xvs


# Expected: All tests pass. If any fail, check:

# - Did you handle variable interpolation correctly? Use the resolver test.

# - Did you correctly parse nested blocks? Check the AST structure.
```

Milestone 2:

```bash
# Run state manager tests
pytest tests/unit/test_state_manager.py -xvs

# Expected: Tests pass, including the atomic write test.
# If locking tests fail on Windows, you may need to adjust file locking implementation.
```

Milestone 3:

```bash
# Run planner tests
pytest tests/unit/test_planner.py -xvs

# Expected: All tests pass, including cycle detection.
# If topological sort fails, verify your graph construction first.
```

Milestone 4:

```bash
# Run provider and executor tests
pytest tests/unit/test_provider.py tests/unit/test_executor.py -xvs

# Expected: Tests pass, including retry and circuit breaker tests.
# If executor tests fail, check that you're calling provider methods in correct order.
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Plan shows no changes when changes exist	Diff computation not detecting attribute changes	Log desired vs. current attributes; check deep equality	Ensure diff compares nested structures, not just top-level dict
Lock never released after crash	Heartbeat thread not stopped; lock file orphaned	Check for stale `.lock` file; verify `LockHandle.cleanup()` called	Implement lock stale detection; add timeout to lock acquisition
Cycle detected in valid config	Over-eager implicit dependency detection	Log extracted references; visualize graph	Limit implicit dependencies to actual attribute references, not all strings
Provider retries exhausting but shouldn't	Non-transient exception being caught	Check exception hierarchy; log exception types	Ensure `TransientProviderError` is raised for retryable errors only
State file corrupted after partial write	Non-atomic write interrupted	Check for `.bak` file; verify `write_atomic_json` renames after write	Use `tempfile.NamedTemporaryFile` with `os.replace` for atomicity
Executor creates resources in wrong order	Topological sort incorrect or graph edges reversed	Print dependency graph and sort order; verify edges point from dependent to dependency	Ensure `depends_on` and implicit refs create edge `dependent -> dependency`

Debugging Guide

> **Milestone(s):** 1, 2, 3, 4 (cross-cutting)

This section provides a comprehensive guide to diagnosing and resolving common issues you'll encounter while implementing the Infrastructure as Code engine. Debugging distributed infrastructure management requires systematic thinking—you need to understand where in the pipeline a failure occurred and how different components interact. Think of debugging as being a **forensic investigator** examining a crime scene: you collect evidence (logs, state files), reconstruct events (execution sequences), and identify the culprit (buggy code or unexpected conditions).

Common Bugs: Symptom → Cause → Fix

The following table catalogs the most frequent issues you'll encounter during implementation, organized by component area. Each entry follows the pattern of observed symptom, root cause, and concrete fix.

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
"Plan shows no changes" when resources should be created	Planner or State Manager	1. **State file corruption**: The state file contains invalid data that doesn't match the parsed resources. 2. **Incorrect diff logic**: `compute_diff` returns `ActionType.NOOP` for resources that differ. 3. **Variable interpolation mismatch**: Variables resolve differently between runs, causing different resource addresses.	1. Examine the state file with `cat state.json` - look for missing or malformed `StateRecord` entries. 2. Add debug logging to `compute_diff` to compare `desired_attributes` vs `current_attributes`. 3. Run `process_configuration` in isolation and inspect the resolved `Resource.attributes`.	1. Use `read_json_with_backup` to restore from backup if corruption is detected. 2. Ensure `compute_diff` compares all attributes, not just a subset. 3. Verify variable resolution produces consistent values across runs.
"Cycle detected" error during plan generation	Planner	1. **Circular dependency**: Resource A depends on B, and B depends on A via explicit `depends_on` or implicit references. 2. **Self-reference**: A resource references its own attribute (e.g., `name = "${self.id}"`). 3. **Transitive cycle**: A → B → C → A forms a three-way cycle.	1. Run `validate_acyclic` with debug output to print the cycle. 2. Inspect the `DependencyGraphNode.depends_on` for each resource in the suspected cycle. 3. Check for implicit dependencies from attribute references that create unintended edges.	1. Review configuration and break the cycle by removing unnecessary `depends_on`. 2. For implicit cycles, consider using `ignore_changes` meta-argument or restructure resources. 3. Implement cycle detection that suggests which dependency to remove.
"Lock never released" - subsequent operations hang	State Manager	1. **Process crash without cleanup**: The previous process acquired a lock but crashed before calling `release_lock`. 2. **Heartbeat thread deadlock**: The `heartbeat_thread` is blocked or not updating the lock timestamp. 3. **Stale lock detection timeout too long**: Default timeout (e.g., 300 seconds) hasn't expired yet.	1. Check lock file timestamp: `stat -c %y .terraform.lock`. 2. Verify if the locking process is still running: `ps -p <process_id>`. 3. Examine lock file contents for `process_id` and `lock_id`.	1. Implement stale lock detection in `acquire_lock` - if lock is older than timeout, forcibly acquire. 2. Ensure `LockHandle.heartbeat_thread` properly updates timestamp with file atomic writes. 3. Add a `force-unlock` CLI command to manually remove stale locks.
"Variable not found" during parsing	Configuration Parser	1. **Variable defined in wrong scope**: Variable defined in child module but referenced in parent. 2. **Variable file not loaded**: `-var-file` not passed or file missing. 3. **Interpolation syntax error**: `${var. name}` has extra space or malformed expression.	1. Print the variable resolution stack trace during `resolve_variables`. 2. List all loaded variable files and CLI variables before resolution. 3. Check raw AST for variable block locations vs reference locations.	1. Ensure variable resolution searches parent scopes for definitions. 2. Implement fallback to environment variables for missing variables. 3. Validate interpolation syntax during parsing, not just resolution.
"Provider not found" when applying plan	Executor/Provider SDK	1. **Provider plugin not installed**: The required provider (e.g., `aws`) isn't in the plugins directory. 2. **Provider configuration missing**: No `provider "aws"` block with required	1. Check `ProviderConfig.provider_type` matches available provider classes. 2. Verify `ProviderConfig.config` contains all required fields. 3. Test credentials independently (e.g., `aws sts get-caller-identity`).	1. Implement provider plugin discovery scanning a `plugins/` directory. 2. Validate provider configuration during parsing, not just at execution. 3. Provide clear error messages including which field is missing.

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
		`region` and `access_key`. 3. **Provider initialization error**: `validate_credentials` fails due to invalid credentials.		
Resource stuck in "Creating..." forever	Executor/Provider	1. **Eventual consistency delay**: Cloud API returns success but resource isn't immediately available. 2. **Missing async polling**: Provider `create` doesn't wait for resource to reach ready state. 3. **Timeout too short**: The polling timeout (e.g., 5 minutes) is less than cloud provisioning time.	1. Check provider logs for API response - look for `pending` or `in-progress` status. 2. Verify `refresh_state` is called after `create` and what state it returns. 3. Monitor cloud console manually to see actual resource status.	1. Implement exponential backoff polling in provider `create` method. 2. Increase timeout for slow resources (databases, VPCs). 3. Add progress indicators showing polling attempts.
Partial apply failure leaves "zombie resources"	Executor	1. **State not updated on failure**: Resource created successfully but error in subsequent step prevents `write_state`. 2. **No rollback on error**: The executor doesn't clean up successfully created resources when later resources fail. 3. **Inconsistent state file**: Partial write due to crash during `write_state`.	1. Compare state file with actual cloud resources using provider `read`. 2. Check for `PartialApplyError.succeeded_resources` to see what was created. 3. Look for `write_atomic_json` temporary files left in directory.	1. Implement two-phase state update: write intermediate state after each successful resource. 2. Add `-auto-approve=false` default to prevent accidental zombie creation. 3. Use backup state file that's only updated after full successful apply.
"Invalid interpolation" in count or for_each	Configuration Parser	1. **Chicken-egg problem**: Variable used in `count` references another resource attribute that doesn't exist yet. 2. **Type mismatch**: `count` expects a number but receives a string or list. 3. **Circular reference in count**: Two resources' counts depend on each other.	1. Evaluate interpolation expressions in isolation to see their resolved value. 2. Check the type of the resolved variable (number vs string). 3. Build dependency graph including count references as edges.	1. Implement two-pass parsing: first resolve static variables, then resolve dynamic ones. 2. Add type validation for `count` and `for_each` arguments. 3. Detect and reject circular count dependencies during graph building.
Graph topological sort returns different order each run	Planner	1. **Non-deterministic iteration over dictionary**: Python's `dict` iteration order affects graph construction. 2. **Missing dependency edge**: Some resources have no dependencies, creating multiple valid topological orders. 3. **Graph nodes not sorted before algorithm**: The algorithm picks arbitrary starting nodes.	1. Print the adjacency list of the graph before sorting. 2. Run topological sort multiple times and compare outputs. 3. Check if all expected dependencies are captured (implicit vs explicit).	1. Sort node IDs alphabetically before starting topological sort. 2. Ensure deterministic edge addition by sorting source and target nodes. 3. Document that multiple valid orders are possible when no dependencies exist.
Rate limit errors	Provider SDK	1. **No jitter in retry delays**: All retries happen	1. Check timestamps of API calls in logs - are they evenly spaced?	1. Add jitter to retry delays using `random.uniform(0.5 * delay,`

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
despite retry logic		simultaneously across parallel requests. 2. **Retry window too short**: Exponential backoff max delay (e.g., 30s) less than rate limit window (e.g., 1 minute). 3. **No circuit breaker**: Repeated failures continue retrying indefinitely.	2. Examine cloud provider rate limit headers (`Retry-After` , `X-RateLimit-Reset`). 3. Monitor circuit breaker state (open/closed/half-open).	`1.5 * delay)` . 2. Respect `Retry-After` headers when present in error responses. 3. Implement circuit breaker that opens after N consecutive failures.
State file grows indefinitely with stale resources	State Manager	1. **No state compaction**: Deleted resources remain in state as `tombstones` or with `ActionType.DELETE` records. 2. **Multiple state versions kept**: Backup mechanism never cleans up old backups. 3. **Resource renaming creates duplicates**: Old `Resource.id` remains while new one is added.	1. Examine state file size over time - does it increase after each apply? 2. Count `StateRecord` entries for deleted resources (attributes null). 3. Check for duplicate resources with similar attributes.	1. Implement state compaction after successful delete operations. 2. Add retention policy for backups (keep last N versions). 3. Add state validation that detects and removes orphaned resources.
Module source path resolution fails	Configuration Parser	1. **Relative path confusion**: Module uses `../parent` but current working directory differs. 2. **Git URL without ref**: Module source is `github.com/org/module` without branch/tag. 3. **Missing module cache**: Module downloaded previously but cache cleared.	1. Print the resolved absolute path for each module source. 2. Check if directory exists at the resolved path. 3. Verify module registry configuration (if using remote modules).	1. Normalize all paths to absolute paths relative to configuration root. 2. Implement module cache with version locking (similar to Terraform's `.terraform/modules`). 3. Provide clear error including the resolved path that failed.

Debugging Techniques and Tools

Effective debugging requires more than just reading error messages—you need a systematic approach to isolate issues. Think of these techniques as **medical diagnostic tools**: each reveals different aspects of the system's health.

Strategic Logging with Context Correlation

The most powerful debugging technique is comprehensive, structured logging. Instead of simple `print` statements, implement a logging system that:

1. **Includes correlation IDs**: Each operation (plan, apply) gets a unique ID logged with every message, making it easy to trace through components.
2. **Uses log levels appropriately**:
   - `DEBUG` : Detailed internal state (attribute values, graph edges)
   - `INFO` : Major lifecycle events (resource created, plan generated)
   - `WARN` : Unexpected but recoverable conditions (falling back to default)
   - `ERROR` : Operation failures with stack traces
3. **Logs to structured formats** (JSON) for machine parsing, with human-readable format for development.

> **Diagnostic Pattern**: When you encounter an error, increase the log level to `DEBUG` , reproduce the issue, and search for the correlation ID in the log file. Follow the execution path through each component.

State File Inspection and Diffing

The state file is the **ground truth** of your infrastructure. Learn to inspect it like a database:

1. **Manual inspection**:

```bash
# Pretty-print the state
jq . terraform.tfstate

# Count resources by type
jq '.resources | group_by(.type) | map({type: .[0].type, count: length})' terraform.tfstate

# Find resources with specific attribute
jq '.resources[] | select(.attributes.name == "web-server")' terraform.tfstate
```

2. **State diffs between runs**:

```bash
# Compare current state with backup
diff -u <(jq -S . terraform.tfstate) <(jq -S . terraform.tfstate.backup)

# Generate visual diff
git diff --no-index terraform.tfstate.old terraform.tfstate.new
```

3. **State validation script**: Create a Python script that loads the state file and validates invariants (no duplicate IDs, all dependencies exist, attributes match schema).

Graph Visualization for Dependency Analysis

When dependency issues arise, visualize the graph to understand relationships:

1. **Export graph to DOT format** for Graphviz:

```python
def export_to_dot(graph: Dict[str, DependencyGraphNode]) -> str:
    lines = ["digraph G {"]
    for node_id, node in graph.items():
        for dep in node.depends_on:
            lines.append(f'  "{dep}" -> "{node_id}";')
    lines.append("}")
    return "\n".join(lines)
```

2. **Render and view**:

```bash
python -c "from planner import build_graph; print(export_to_dot(graph))" > graph.dot
dot -Tpng graph.dot -o graph.png
open graph.png  # Or display on Linux
```

3. **Interactive visualization** with network libraries (vis.js, D3.js) for complex graphs with 50+ nodes.

Stepwise Execution with Intermediate Dumps

When an algorithm fails (like topological sort), capture its intermediate state:

1. **Add checkpoint debugging** that saves state at key points:

```python
def topological_sort_with_debug(graph):
    checkpoint = 1
    while nodes_with_zero_indegree:
        save_checkpoint(checkpoint, {"graph": graph.copy(), "sorted": sorted_nodes})
        checkpoint += 1
        # ... algorithm continues
```

2. **Create a replay capability** that loads checkpoint data and continues from that point, allowing you to test fixes without re-running the entire pipeline.

Mock Provider for Isolated Testing

When debugging provider issues, use a mock provider that simulates cloud APIs:

1. **In-memory mock** that stores resources in a dictionary, simulating eventual consistency with configurable delays.
2. **Fault-injection mock** that fails specific operations (create, update) at certain rates to test error handling.
3. **Record/replay mock** that captures real API calls during one run and replays them during tests, eliminating cloud dependencies.

Time Travel Debugging with State Snapshots

For intermittent issues, implement a **state snapshot** system:

1. **Automatic snapshots** before each state-modifying operation.
2. **Label snapshots** with metadata (timestamp, operation ID, user).
3. **Restore capability** to roll back to any snapshot:

```bash
iac-engine state restore --snapshot-id plan-2023-10-05-14-30-00
```

Implementation Guidance

This section provides concrete tools and code patterns to implement the debugging techniques described above.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging	Python `logging` module with JSON formatter	Structured logging with `structlog` or `loguru` for context propagation
State Inspection	`jq` command-line tool with Python wrapper	Custom web UI with React + D3 for visual state exploration
Graph Visualization	Graphviz DOT format generation	Interactive visualization with `pyvis` or `networkx` + `matplotlib`
Debug UI	CLI with rich tables via `rich` or `textual`	Web dashboard with FastAPI + WebSocket for real-time updates
Snapshot Management	Directory of timestamped JSON files	Dedicated versioned store with content-addressable hashing

Recommended File Structure for Debugging Utilities

```
iac-engine/
  src/
    iac/
      debug/                    # Debugging utilities
        __init__.py
        logger.py               # Structured logging setup
        state_inspector.py      # State file analysis tools
        graph_exporter.py       # Graph visualization exports
        snapshot_manager.py     # State snapshot management
      cli/
        commands/
          debug.py              # Debug-specific CLI commands
```

Infrastructure Starter Code: Structured Logger

PYTHON

```python
# src/iac/debug/logger.py

import json

import logging

import uuid

from datetime import datetime

from typing import Any, Dict, Optional

from contextvars import ContextVar


# Correlation ID for tracing requests across components

correlation_id: ContextVar[str] = ContextVar('correlation_id', default='')


class JsonFormatter(logging.Formatter):

    """Formats log records as JSON for machine parsing."""



    def format(self, record: logging.LogRecord) -> str:

        log_object = {

            'timestamp': datetime.utcfromtimestamp(record.created).isoformat() + 'Z',

            'level': record.levelname,

            'logger': record.name,

            'message': record.getMessage(),

            'correlation_id': correlation_id.get(),

            'module': record.module,

            'function': record.funcName,

            'line': record.lineno,

        }



        # Add extra fields if present

        if hasattr(record, 'extra'):

            log_object.update(record.extra)



        # Add exception info if present

        if record.exc_info:

            log_object['exception'] = self.formatException(record.exc_info)



        return json.dumps(log_object)


def setup_logging(level: str = 'INFO', json_format: bool = False) -> None:

    """Configure logging for the IaC engine."""

    logger = logging.getLogger('iac')
```

```
    logger.setLevel(getattr(logging, level.upper()))

    handler = logging.StreamHandler()

    if json_format:
        handler.setFormatter(JsonFormatter())
    else:
        # Human-readable format for development
        formatter = logging.Formatter(
            '%(asctime)s [%(correlation_id)s] %(levelname)s %(name)s: %(message)s',
            datefmt='%H:%M:%S'
        )
        handler.setFormatter(formatter)

    logger.addHandler(handler)

    # Set correlation ID filter
    class CorrelationFilter(logging.Filter):
        def filter(self, record: logging.LogRecord) -> bool:
            record.correlation_id = correlation_id.get()
            return True

    logger.addFilter(CorrelationFilter())

def log_operation(operation: str, **extra: Any) -> str:
    """Start a new operation with a correlation ID and log its beginning."""
    op_id = str(uuid.uuid4())[:8]
    correlation_id.set(op_id)

    logger = logging.getLogger('iac')
    logger.info(f"Starting {operation}", extra={'operation': operation, **extra})

    return op_id

# Usage example:
# from iac.debug.logger import setup_logging, log_operation
# setup_logging(level='DEBUG')
# op_id = log_operation('plan', config_path='main.tf')
# # All subsequent logs will include this correlation_id
```

Infrastructure Starter Code: State Inspector

```python
# src/iac/debug/state_inspector.py

import json

from pathlib import Path

from typing import Dict, List, Any, Optional

from dataclasses import dataclass

from iac.state import StateRecord


@dataclass

class StateReport:

    """Comprehensive analysis of a state file."""

    total_resources: int

    resources_by_type: Dict[str, int]

    orphaned_resources: List[str]  # Resources with missing dependencies

    duplicate_addresses: List[str]  # Multiple resources with same address

    validation_errors: List[str]



    def to_dict(self) -> Dict[str, Any]:

        return {

            'total_resources': self.total_resources,

            'resources_by_type': self.resources_by_type,

            'orphaned_resources': self.orphaned_resources,

            'duplicate_addresses': self.duplicate_addresses,

            'validation_errors': self.validation_errors,

            'is_valid': len(self.validation_errors) == 0

        }


class StateInspector:

    """Tools for analyzing and debugging state files."""



    @staticmethod

    def load_state(state_path: Path) -> Dict[str, Any]:

        """Load state file with backup fallback."""

        from iac.state import read_json_with_backup

        return read_json_with_backup(state_path)



    @staticmethod

    def analyze(state_data: Dict[str, Any]) -> StateReport:

        """Perform comprehensive analysis of state data."""

        resources = state_data.get('resources', [])
```

```python
# Count by type
resources_by_type: Dict[str, int] = {}
for resource in resources:
    rtype = resource.get('type', 'unknown')
    resources_by_type[rtype] = resources_by_type.get(rtype, 0) + 1

# Check for duplicates by address
addresses: Dict[str, int] = {}
duplicate_addresses: List[str] = []
for resource in resources:
    addr = f"{resource.get('type')}.{resource.get('name')}"
    addresses[addr] = addresses.get(addr, 0) + 1
    if addresses[addr] > 1:
        duplicate_addresses.append(addr)

# Find orphaned resources (dependencies that don't exist)
orphaned_resources: List[str] = []
all_resource_addrs = {f"{r.get('type')}.{r.get('name')}" for r in resources}

for resource in resources:
    deps = resource.get('dependencies', [])
    for dep in deps:
        if dep not in all_resource_addrs:
            orphaned_resources.append(
                f"{resource.get('type')}.{resource.get('name')} -> {dep}"
            )

# Collect validation errors
validation_errors: List[str] = []

if duplicate_addresses:
    validation_errors.append(
        f"Duplicate resource addresses: {', '.join(set(duplicate_addresses))}"
    )

if orphaned_resources:
    validation_errors.append(
        f"Resources with missing dependencies: {len(orphaned_resources)}"
```

```
            )

        return StateReport(
            total_resources=len(resources),
            resources_by_type=resources_by_type,
            orphaned_resources=orphaned_resources,
            duplicate_addresses=list(set(duplicate_addresses)),
            validation_errors=validation_errors
        )

    @staticmethod
    def compare(state_a: Dict[str, Any], state_b: Dict[str, Any]) -> Dict[str, Any]:
        """Compare two state files and return differences."""
        resources_a = {f"{r['type']}.{r['name']}": r for r in state_a.get('resources', [])}
        resources_b = {f"{r['type']}.{r['name']}": r for r in state_b.get('resources', [])}

        all_addrs = set(resources_a.keys()) | set(resources_b.keys())

        diff = {
            'added': [],
            'removed': [],
            'modified': [],
            'unchanged': []
        }

        for addr in all_addrs:
            if addr in resources_a and addr not in resources_b:
                diff['removed'].append(addr)
            elif addr not in resources_a and addr in resources_b:
                diff['added'].append(addr)
            elif resources_a[addr] == resources_b[addr]:
                diff['unchanged'].append(addr)
            else:
                # Deep compare attributes
                attrs_a = resources_a[addr].get('attributes', {})
                attrs_b = resources_b[addr].get('attributes', {})

                # Simple attribute comparison (could be enhanced)
                if attrs_a != attrs_b:
```

```
                    diff['modified'].append({
                        'address': addr,
                        'attributes_changed': True
                    })

        return diff

# CLI command to inspect state
def inspect_state_command(state_path: Path, output_format: str = 'text') -> None:
    """CLI command for state inspection."""
    from iac.debug.state_inspector import StateInspector, StateReport

    state_data = StateInspector.load_state(state_path)
    report = StateInspector.analyze(state_data)

    if output_format == 'json':
        import json
        print(json.dumps(report.to_dict(), indent=2))
    else:
        print(f"State File: {state_path}")
        print(f"Total Resources: {report.total_resources}")
        print("\nResources by Type:")
        for rtype, count in report.resources_by_type.items():
            print(f"  {rtype}: {count}")

        if report.duplicate_addresses:
            print(f"\n⚠️  Duplicate Addresses: {len(report.duplicate_addresses)}")
            for addr in report.duplicate_addresses[:5]:  # Show first 5
                print(f"  {addr}")

        if report.orphaned_resources:
            print(f"\n⚠️  Orphaned Dependencies: {len(report.orphaned_resources)}")
            for orphan in report.orphaned_resources[:5]:
                print(f"  {orphan}")

        if report.validation_errors:
            print(f"\n❌ Validation Errors:")
            for error in report.validation_errors:
                print(f"  {error}")
```

```
    else:
        print("\n✅ State file appears valid")
```

Core Logic Skeleton: Graph Visualization Exporter

```python
# src/iac/debug/graph_exporter.py

from typing import Dict, List, Optional

from iac.planner import DependencyGraphNode


class GraphExporter:

    """Exports dependency graphs to various visualization formats."""


    @staticmethod

    def to_dot(graph: Dict[str, DependencyGraphNode],

               highlight_cycles: Optional[List[List[str]]] = None) -> str:

        """Export graph to Graphviz DOT format.


        Args:

            graph: Dependency graph from build_graph

            highlight_cycles: List of cycles to highlight in red


        Returns:

            DOT format string for rendering with Graphviz

        """

        # TODO 1: Initialize DOT string with digraph header and styling options

        # Example: 'digraph G { rankdir=TB; node [shape=box];'


        # TODO 2: Add all nodes with labels

        # For each node_id in graph.keys():

        #   Create node with label like "type.name"

        #   Format: '  "aws_instance.web" [label="aws_instance.web"];'


        # TODO 3: Add all edges for dependencies

        # For each node_id, node in graph.items():

        #   For each dep in node.depends_on:

        #     Add edge: '  "aws_vpc.main" -> "aws_instance.web";'


        # TODO 4: If highlight_cycles provided, color cycle nodes and edges

        # For each cycle in highlight_cycles:

        #   For node_id in cycle:

        #     Add node attribute: '  "node_id" [color=red, penwidth=2];'

        #   For edges within cycle:

        #     Add edge attribute: '  "a" -> "b" [color=red, penwidth=2];'
```

```python
        # TODO 5: Close the graph and return complete DOT string
        pass

    @staticmethod
    def to_mermaid(graph: Dict[str, DependencyGraphNode]) -> str:
        """Export graph to Mermaid.js format for web display.

        Returns:
            Mermaid graph definition string
        """
        # TODO 1: Start with 'graph TD' for top-down layout

        # TODO 2: Add all nodes with sanitized IDs
        # Mermaid IDs can't contain dots, so convert 'aws_instance.web' to 'aws_instance_web'

        # TODO 3: Add all edges using Mermaid arrow syntax
        # Format: 'aws_vpc_main --> aws_instance_web;'

        # TODO 4: Add click handlers for interactive debugging
        # Format: 'click aws_instance_web callback "show_resource_details"'

        pass

    @staticmethod
    def find_cycles(graph: Dict[str, DependencyGraphNode]) -> List[List[str]]:
        """Find all cycles in the graph using DFS.

        Returns:
            List of cycles, where each cycle is a list of node IDs
        """
        # TODO 1: Initialize visited set, recursion stack, and cycle list

        # TODO 2: Implement recursive DFS with cycle detection
        # Use standard algorithm: white/gray/black sets for unvisited/visiting/visited

        # TODO 3: When back edge found (node in gray set), reconstruct cycle path

        # TODO 4: Return list of unique cycles (avoid duplicates)
```

```
    pass
```

Debug CLI Command Skeleton

```python
# src/iac/cli/commands/debug.py

import click

from pathlib import Path

from typing import Optional


@click.group(name="debug")

def debug_cli():

    """Debugging commands for the IaC engine."""

    pass


@debug_cli.command("state")

@click.argument("state_path", type=click.Path(exists=True, path_type=Path))

@click.option("--format", type=click.Choice(["text", "json", "html"]), default="text")

def state_inspect(state_path: Path, format: str):

    """Inspect and validate a state file."""

    from iac.debug.state_inspector import inspect_state_command

    inspect_state_command(state_path, format)


@debug_cli.command("graph")

@click.argument("config_path", type=click.Path(exists=True, path_type=Path))

@click.option("--output", "-o", type=click.Path(path_type=Path), default="graph.png")

@click.option("--format", type=click.Choice(["dot", "mermaid", "png", "svg"]), default="png")

def graph_export(config_path: Path, output: Path, format: str):

    """Export dependency graph to visualization format."""

    # TODO 1: Parse configuration using process_configuration


    # TODO 2: Load state if exists


    # TODO 3: Build graph using build_graph


    # TODO 4: Export using GraphExporter based on format


    # TODO 5: If format is png/svg, call Graphviz to render

    #   import subprocess

    #   subprocess.run(["dot", "-Tpng", "graph.dot", "-o", output])


    click.echo(f"Graph exported to {output}")


@debug_cli.command("replay")

@click.argument("snapshot_id")
```

```python
@click.option("--step", "-s", type=int, help="Execute only up to this step")
def replay_snapshot(snapshot_id: str, step: Optional[int]):
    """Replay execution from a saved snapshot."""
    # TODO 1: Load snapshot by ID from snapshot manager


    # TODO 2: Restore state to snapshot state


    # TODO 3: If step provided, execute only up to that step


    # TODO 4: Provide interactive prompt to continue or inspect
    click.echo(f"Replaying snapshot {snapshot_id}")

# Register with main CLI
# In main cli.py: cli.add_command(debug_cli)
```

Language-Specific Hints for Python

1. **Use `pdb` or `ipdb` for interactive debugging**:

```python
import ipdb; ipdb.set_trace()  # Add breakpoint
# Then inspect variables, step through code
```

2. **Leverage `inspect` module for runtime introspection**:

```python
import inspect
print(inspect.getsource(compute_diff))  # View function source
print(inspect.signature(plan_command))  # View parameter signature
```

3. **Use `traceback` for better error context**:

```python
import traceback
try:
    apply_plan(plan, providers)
except Exception as e:
    traceback.print_exc()  # Full stack trace
    logger.error("Apply failed", exc_info=True)  # Include in logs
```

4. **Profile performance bottlenecks**:

```python
import cProfile
pr = cProfile.Profile()
pr.enable()
plan_command(config_path, state_path)
pr.disable()
pr.print_stats(sort='cumtime')
```

Milestone Checkpoint: Debugging Readiness

After implementing the debugging utilities, verify they work:

```bash
# Test structured logging

python -c "from iac.debug.logger import setup_logging; setup_logging('DEBUG'); import logging; logging.getLogger('iac').info('Test log')"


# Test state inspection

python src/iac/cli/main.py debug state terraform.tfstate --format=json


# Test graph export (requires Graphviz installed)

python src/iac/cli/main.py debug graph main.tf --output=dependency.png


# Expected outputs:

# - Logs appear in JSON format with correlation IDs

# - State inspection shows resource counts and validation results

# - dependency.png file created showing visual graph
```

Signs something is wrong:

- Logs don't appear: Check logging level and handler configuration
- State inspection crashes: State file may be malformed - add try/except
- Graph export fails: Graphviz may not be installed (`brew install graphviz` or `apt-get install graphviz`)

Remember: debugging is iterative. Start with logging, then add visualization, then build interactive tools as needed. The goal is to make the system's internal state as transparent as possible.

Future Extensions

> **Milestone(s):** This section describes potential enhancements that build upon the core system and are relevant to all milestones (1, 2, 3, 4).

The foundational IaC engine we've designed provides a robust, extensible platform for managing infrastructure declaratively. By adhering to clean abstractions—particularly the separation between configuration, planning, and execution—the architecture can accommodate numerous enhancements without requiring fundamental redesign. This section explores several valuable extensions that could be built upon the existing components, each addressing a real-world operational need that mature IaC tools typically provide.

Possible Enhancements

The following enhancements represent natural evolution paths for the system, organized by the operational challenge they solve.

Validation Webhook (Pre-Apply Policy Enforcement)

Mental Model: The Building Inspector

> Think of this enhancement as adding a building inspector who must approve the blueprints (execution plan) before any construction begins. The inspector reviews the plan against a set of safety codes (policies) and can either grant a permit (allow the apply) or require revisions (reject the plan).

A validation webhook allows external systems to inspect and approve or reject an execution plan before it's applied. This is crucial for enforcing organizational policies (e.g., "no S3 buckets can be publicly readable," "all compute instances must have specific tags") in an automated, non-bypassable way.

Component	Modification Required	Description
`Executor`	New `validate_plan` method	Instead of proceeding directly to `apply_plan`, the executor would first serialize the plan and send it to a configured webhook URL.
`PlanAction`	None (serialization already exists via `JsonSerializable`)	The plan's `to_dict()` or `to_json()` would be used to create the payload.
New: `WebhookValidator`	Entire new component	Handles HTTP communication, retries, timeout, and response parsing. Integrates with the `Executor`.

Architecture Decision: Synchronous vs. Asynchronous Validation

> **Decision: Synchronous Validation with Timeout**
>
> - **Context**: We need to guarantee policy enforcement before any resources are modified. The apply operation must wait for a definitive allow/deny decision.
> - **Options Considered**:
>   1. **Synchronous HTTP call**: The `apply_command` blocks, sends the plan to the webhook, and proceeds only after receiving an `APPROVE` response.
>   2. **Asynchronous approval ticket**: The system creates a ticket (e.g., in a queue or issue tracker) and pauses. A separate human or automated process reviews and approves later, after which the apply resumes.
> - **Decision**: Implement synchronous validation for fully automated pipelines, with a configurable timeout (e.g., 30 seconds).
> - **Rationale**: Synchronous validation provides immediate feedback and fits seamlessly into CI/CD pipelines where the entire workflow must pass or fail in a single run. The timeout prevents the system from hanging indefinitely if the webhook is unresponsive.
> - **Consequences**: Adds a critical dependency on the webhook service's availability. A down webhook will block all applies unless the feature is explicitly disabled. This can be mitigated with circuit breaker patterns in the `WebhookValidator`.

Integration Point: The `apply_command` would be extended with a new step between acquiring the state lock and beginning execution. A high-level algorithm would be:

1. Generate the execution plan as usual.
2. If a webhook URL is configured in the environment or config: a. Serialize the plan and current state snapshot. b. POST to the webhook with a correlation ID. c. Wait for response (with timeout and retry for network errors). d. If response contains `{"result": "REJECTED", "reason": "..."}`, abort the apply, release the lock, and output the rejection reason. e. If response is `{"result": "APPROVED"}`, proceed to apply.
3. If the webhook times out or returns an unexpected error, follow a configurable policy (fail open or fail closed).

Policy-as-Code Integration

Mental Model: The Automated Rulebook

> This enhancement embeds a rulebook (policy engine) directly into the planning phase. Imagine a rulebook that automatically flags any blueprint line that violates company policy, similar to a spell-checker for infrastructure configurations.

While a validation webhook is external, Policy-as-Code integrates policy evaluation directly into the engine, typically as a library (like Open Policy Agent - OPA). Policies are written in a dedicated language and evaluate the planned changes against rules that can be more complex than simple webhook checks (e.g., "if the resource is in production, it must have at least two replicas").

Integration Point	Design Approach
Policy Evaluation Time	During planning, after `generate_plan` but before output. Could also run during parsing for early syntax validation.
Policy Scope	Can evaluate the entire config (`List[Resource]`), the computed diff (`Dict[str, PlanAction]`), or individual resources.
Policy Language	Embed a generic engine (e.g., OPA Rego) or define a custom DSL. The `BaseProvider` could also contribute resource-specific validation schemas.

Common Pitfalls and Solutions:

- ⚠️ **Pitfall: Performance degradation on large plans**.
  - **Why it's wrong**: Evaluating complex policies against hundreds of resources can make the `plan` command unacceptably slow.

    - **Fix**: Implement policy filtering (only run certain policies on certain resource types) and caching of policy query results. Consider a parallel evaluation mode.
- ⚠️ **Pitfall: Policy rules conflict with provider validation**.
    - **Why it's wrong**: A policy might reject a configuration that the cloud provider would actually accept, causing confusion.
    - **Fix**: Clearly document that policies are for *organizational* guardrails, not syntax validation. Layer policies: syntax (provider schema) → semantics (policy) → runtime (webhook).

Example Workflow Addition: A new `validate_policies` function could be added to the `Planner` component or as a separate `PolicyEngine` class. The `plan_command` would then:

```
1. Parse config -> desired resources.
2. Read state.
3. Build graph, compute diff, generate plan (as before).
4. Call `policy_engine.validate(plan)`.
5. If violations found, print them as warnings or errors (based on severity config) and optionally exit with a non-zero code.
6. Continue to display plan.
```

PLAINTEXT

This keeps the policy evaluation inside the planning phase, giving users immediate feedback.

Drift Detection Scheduler

Mental Model: The Periodic Inventory Auditor

> Imagine a warehouse manager who, every night, compares the official inventory log (state file) against a physical walkthrough of the shelves (actual cloud resources). Any discrepancies (drift) are recorded in a report for the morning shift.

Drift detection is the process of periodically comparing the recorded state in the state file against the actual state in the cloud, identifying any changes made outside of the IaC tool (e.g., a console edit, CLI command, or another automation tool). Our architecture inherently supports this because the `Provider.read` method already knows how to fetch current state, and `compute_diff` can compare it to the recorded state.

New Component	Purpose
`DriftDetector`	Orchestrates periodic refresh and diff. Could be a long-running daemon or a cron-triggered script.
`DriftReport`	Extends `StateReport` with fields like `drifted_resources: List[Tuple[str, Dict[str, Any]]]` showing the resource address and the attributes that differ.
Notification Integrations	Plugins to send drift reports via email, Slack, or incident management systems.

Design Considerations:

1. **Detection Cadence**: Should be configurable per-resource or per-provider. Some resources change frequently (auto-scaling group instance count) and might be ignored.
2. **State Refresh Strategy**: The `refresh_state` function (part of the executor) can be reused. However, a drift detection run should be *read-only* and must not acquire the state lock (or acquire it in shared read mode if implemented).
3. **Handling Drift**: The system could generate an alert, automatically create a corrective plan, or even apply it (in auto-remediate mode). The safest initial implementation is alert-only.

Algorithm for Drift Detection Run:

1. For each provider configured, instantiate the provider with read-only credentials if possible.
2. Load the state file (read-only, no lock).
3. For each `StateRecord` in state: a. Call `provider.read(resource_id, resource_type)`. b. Compare returned attributes with `StateRecord.attributes`. c. If any differences exist (ignoring certain metadata fields like timestamps), mark as drifted.
4. Generate a `DriftReport` and trigger configured notifications.
5. Optionally, if `auto_remediate: true`, call `plan_command` and `apply_command` with the current state and config to converge.

Integration with Existing Components: The `StateInspector` component (from our debugging guide) could be extended with a `detect_drift` method, leveraging the `compute_diff` function but with the "desired" state being the recorded state and the "current" state being the freshly read cloud state. A `drift_command` CLI command could be added.

Workspace and Environment Management

Mental Model: The Construction Site Blueprint Copies

> Imagine you have one blueprint (configuration), but you need to build the same structure on three different construction sites (environments: dev, staging, prod). Each site has its own log of what's been built (state file) and slightly different characteristics (variables like size, location). The site manager needs to keep these contexts separate but use the same underlying plans.

Our current design assumes a single state file path. In practice, teams manage multiple environments (dev/staging/prod) or multiple logical groupings (workspaces). This enhancement adds a layer of isolation, allowing the same configuration to be applied with different variable values and state files.

Concept	Implementation Approach
Workspace	A named container for a distinct state file (e.g., `terraform.tfstate.d/dev/`). The workspace name becomes a prefix or suffix for the state file path.
Environment Variables	Variable resolution would prioritize workspace-specific variable files (e.g., `dev.tfvars`) over generic ones.
State Isolation	The `StateManager` would be extended to accept a `workspace` parameter, which it uses to construct the state and lock file paths (e.g., `terraform.tfstate.d/{workspace}/terraform.tfstate`).

Required Changes:

- **CLI Commands**: Add `workspace list`, `workspace new`, `workspace select` commands to manage workspaces.
- **State Manager**: Modify `read_state`, `write_state`, and `acquire_lock` to incorporate a workspace-aware path.
- **Parser**: Extend variable resolution to load workspace-specific var files.

Architecture Decision: State File Organization

> **Decision: Directory-Based Workspace Isolation**
>
> - **Context**: We need to store multiple state files for the same configuration root without conflict.
> - **Options Considered**:
>   1. **Directory per workspace**: `state.d/<workspace>/terraform.tfstate`.
>   2. **State file with prefix/suffix**: `terraform-<workspace>.tfstate` in the same directory.
>   3. **Database backend with workspace column**: Store all states in a single database table, keyed by config path + workspace.
> - **Decision**: Use a directory-based approach (`state.d/<workspace>/`).
> - **Rationale**: Simplicity and compatibility with file-based backup/restore tools. It mirrors Terraform's approach, making it familiar to users. Also, it works seamlessly with both local and remote (S3) backends using path prefixes.
> - **Consequences**: Requires filesystem support for directories. Remote backends must support path prefixes (S3 does). Workspace switching becomes a matter of changing the current working directory or setting an environment variable.

Advanced State Backends and Encryption

Mental Model: The Bank Vault Upgrade

> Initially, the ledger (state file) is kept in a filing cabinet (local disk). This enhancement moves it to a secure, replicated, access-controlled vault (remote backend with encryption and versioning).

While Milestone 2 includes a remote backend (e.g., S3), advanced backends could include:

- **Versioned State**: Every `write_state` creates a new version with a commit-like message, enabling rollback to previous states.
- **State Encryption**: Encrypt the state file at rest using customer-managed keys (CMK) or a hardware security module (HSM) integration.
- **State Query API**: Expose a read-only HTTP API for other tools to query the current state (e.g., "list all EC2 instance IPs").

Design Extension: The `StateManager` would use a backend interface, similar to the `BaseProvider` pattern. Different backends (local, S3, HTTP API, database) would implement this interface. Encryption could be a wrapper backend that encrypts/decrypts data before passing it to the underlying backend.

```
interface StateBackend:
    read_state(workspace: str) -> Dict[str, StateRecord]
    write_state(workspace: str, state_data: Dict[str, StateRecord], version_metadata: Optional[Dict]) -> None
    lock_state(workspace: str, info: Dict) -> LockHandle
    unlock_state(workspace: str, lock_handle: LockHandle) -> None
```
PLAINTEXT

Integration Example: To add versioning, the `write_state` method would accept optional metadata (like a user, timestamp, and comment). The S3 backend could use S3 versioning, and also store metadata in a separate manifest file. The `state_inspect_command` could then list versions and show diffs between

them.

Resource Import and State Surgery

Mental Model: The Archaeologist's Catalog

> When discovering ancient artifacts already in the ground, an archaeologist doesn't build them—they carefully catalog them and bring them into the museum's inventory system. Similarly, this feature allows bringing existing cloud resources under IaC management.

Resource import is the process of taking an existing cloud resource that wasn't created by the IaC engine and adding it to the state file, so that future plans can manage it. This requires:

1. Identifying the resource's cloud-side ID and attributes.
2. Matching it to a resource block in the configuration.
3. Writing a `StateRecord` for it without performing a create operation.

How the Architecture Supports It: The `Provider.read` method already fetches current attributes. We would add a new `import_resource` method to the `BaseProvider` interface that takes a resource address and cloud ID, reads the resource, and returns a `StateRecord`. The `StateManager` would then write this record. The tricky part is ensuring the configuration matches; a dry-run or validation step is needed.

New CLI Command: `import_command <resource_address> <cloud_id>` would:

1. Parse config to find the resource definition.
2. Call `provider.import_resource(resource_address, cloud_id)`.
3. Add the returned `StateRecord` to the state (with a lock).
4. Output a summary.

This feature highlights the power of the clear separation between state and configuration: import only manipulates state, leaving the configuration unchanged.

Advanced Execution Strategies

Mental Model: The Traffic Controller for Construction Crews

> Instead of sending all construction crews to the site at once, a traffic controller staggers their arrival, monitors for accidents, and can reroute or pause new crews if problems arise.

Our current `Executor` applies the plan in dependency order with basic concurrency. Advanced strategies could include:

- **Canary Apply**: Apply changes to a small subset of resources first (e.g., one availability zone), verify health, then proceed to the rest.
- **Blue-Green Deployment**: For resources like load balancers or ASGs, create a new parallel set of resources (green), switch traffic, then delete the old (blue).
- **Automatic Rollback on Failure**: If a certain percentage of resources fail during apply, automatically revert the changes by applying the previous state.

Design Implications: These strategies require the `Executor` to have more sophisticated control flow, potentially grouping resources into stages and monitoring outcomes. The `PlanAction` might need additional metadata (like resource tags for canary grouping). The `apply_plan` algorithm would evolve from a simple loop to a state machine orchestrating stages.

While these strategies are complex, the foundational building blocks—dependency graph, plan generation, and provider CRUD—remain the same. The enhancement would be a new `StrategicExecutor` that wraps or extends the basic `Executor`.

Summary of Architectural Flexibility

The table below summarizes how the core architecture's design decisions enable these extensions:

Architectural Feature	Enabled Extensions	Rationale
Provider Plugin Interface (`BaseProvider`)	New cloud services, policy engines, state backends	The interface abstraction allows new capabilities to be added via plugins without modifying core engine code.
Explicit State Serialization (`StateRecord` , `JsonSerializable`)	Drift detection, versioning, import, webhook validation	The state is a serializable, well-defined data structure that can be easily read, compared, and transmitted.
Declarative Resource Model (`Resource` vs. `StateRecord`)	Policy-as-code, visualization, advanced planning	The clear separation between desired (config) and actual (state) enables analysis and transformation at multiple points.
Dependency Graph as First-Class Construct (`DependencyGraphNode`)	Execution strategies, visualization, impact analysis	The explicit graph structure can be traversed, filtered, and displayed for various operational needs.
Modular Component Design (Parser, State, Planner, Executor)	Workspace management, scheduler integration	Components have clean interfaces, allowing them to be wrapped or extended independently (e.g., a scheduler that calls the parser and planner).

> **Key Insight**: The most valuable architectural decision for extensibility was enforcing a unidirectional data flow: Configuration → Resources → Plan → Execution → Updated State. This pipeline model allows "middleware" components (like policy checkers, webhooks, or drift detectors) to intercept and process data at clear stages without side-effects on other stages.

By building upon these solid foundations, the IaC engine can evolve from a simple configuration applier to a comprehensive infrastructure governance platform, all while maintaining the core principles of safety, idempotency, and declarative intent that make IaC powerful.

Glossary

> **Milestone(s):** This section defines key terminology used throughout the design document and is relevant to all milestones (1, 2, 3, 4).

A shared vocabulary is essential for clear communication about a complex system. This glossary defines the key terms, acronyms, and domain-specific language used throughout this design document and the Infrastructure as Code (IaC) engine implementation.

Term Definitions

Term	Definition	Related Concepts & Notes
`ActionType`	An enumeration representing the four fundamental operations the engine can perform on a resource: `CREATE`, `UPDATE`, `DELETE`, and `NOOP`.	**`PlanAction`** objects have an `action_type` field of this type. `NOOP` indicates the resource already matches the desired state.
`ApplyResult`	A data structure (`ApplyResult`) returned by the executor for each resource action attempted during an apply operation. It records success/failure, the resulting state, any errors, and retry counts.	Used for reporting and handling **partial apply** failures. Contains fields: `resource_address`, `success`, `new_state`, `error`, `retries`.
Atomic Operation	An operation that either completes fully or not at all, with no observable intermediate state. In state management, **atomic file writes** (via rename) prevent **state corruption** from partial writes.	Contrasts with non-atomic operations. Implemented via `write_atomic_json`.
BaseProvider	The core interface (`BaseProvider`) that all cloud provider plugins must implement. It defines the standard CRUD operations (`create`, `read`, `update`, `delete`) and configuration validation (`validate_credentials`).	Enables **polymorphism**; the **Executor** interacts with all providers through this interface.
Circuit Breaker	A resilience pattern that stops calling a failing service after a threshold of failures is reached, allowing it time to recover. It fails fast (**fail-fast**) instead of wasting resources on likely failures.	Implemented by the `CircuitBreaker` class. When open, calls result in a `CircuitOpenError`. Helps manage **rate limiting** and **transient provider errors**.
ConfigurationError	A subclass of `IaCError` raised when there is a problem with the user's configuration files, such as invalid syntax, undefined variables, or invalid resource schemas.	Handled during the parsing and validation phase (**Milestone 1**).
Correlation ID	A unique identifier attached to a single operation (e.g., an `apply` command) and included in all logs and requests related to that operation. Enables tracing the flow of execution across components for debugging.	Part of **structured logging** via `log_operation`.
CRUD	Acronym for **Create, Read, Update, Delete**—the four basic operations of persistent storage. The `BaseProvider` interface is a CRUD interface for cloud resources.	
DAG (Directed Acyclic Graph)	A **directed graph** with no cycles, used to model resource dependencies. The **Planner** builds a DAG where nodes are resources and edges represent "depends on" relationships.	**Topological sort** requires a DAG. **Cycle detection** is performed to ensure the graph is acyclic.
Declarative Configuration	An approach where the user specifies the *desired end state* of the infrastructure (what resources should exist and their properties), not the step-by-step commands to achieve it. The IaC engine's job is to figure out and execute the necessary changes.	Contrasts with *imperative* or *procedural* approaches (e.g., scripts). Central philosophy of this project.
Dependency Graph	A graph representation of resource dependencies, built by the **Planner**. Implemented as a collection of `DependencyGraphNode` objects.	Used to determine the correct order of operations during `apply`.
`DependencyGraphNode`	A data structure representing a single resource within the **dependency graph**. It contains the resource's unique address and lists of resources it `depends_on` and is `required_by`.	Fields: `resource_id`, `depends_on`, `required_by`.
Diff	The comparison between the **desired state** (derived from configuration) and the **current state** (from the state file) for a specific resource. The result of this comparison determines the `PlanAction` (`CREATE`, `UPDATE`, `DELETE`, `NOOP`).	Computed by `compute_diff`.
Directed Acyclic Graph (DAG)	See **DAG**.	
Eventual Consistency	A property of many cloud APIs where a change (e.g., creating a resource) is not immediately reflected in all subsequent read requests. The system guarantees consistency only after an undefined period.	A major challenge for providers; requires robust **retry logic** and **state refresh**.

Term	Definition	Related Concepts & Notes
Execution Plan	The final output of the `plan` command: an ordered list of `PlanAction` objects specifying the precise create, update, and delete operations needed to converge current state to desired state.	The "what will happen" preview shown to the user before `apply`.
Executor	The component (**Component Design: Provider Abstraction & Executor**) responsible for carrying out an **execution plan**. It orchestrates calls to **providers**, handles **concurrency**, **retries**, and updates the **state**.	Calls `apply_plan`.
Exponential Backoff	A retry strategy where the delay between retries increases exponentially (e.g., 1s, 2s, 4s, 8s). Often combined with **jitter** to prevent synchronized retry storms from multiple clients.	Implemented via the `retry_with_backoff` decorator for handling **transient provider errors**.
Fail-fast	A design approach where a system immediately reports an error condition rather than attempting to proceed in an invalid or degraded state. The **circuit breaker** pattern is a form of fail-fast for downstream service failures.	
Graph Visualization	The process of generating a visual diagram (e.g., DOT, Mermaid) of the **dependency graph** to aid in understanding and debugging resource relationships.	The `GraphExporter` class provides methods like `to_dot` and `to_mermaid`.
HCL (HashiCorp Configuration Language)	A declarative configuration language created by HashiCorp, used by Terraform. It is a **DSL** designed for defining infrastructure. This project parses an HCL-like syntax.	
IaC (Infrastructure as Code)	The practice of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.	The overarching domain of this project.
`IaCError`	The base exception class for all errors thrown by the IaC engine. Specific error types like `ConfigurationError`, `StateError`, and `ProviderError` inherit from it.	Allows for centralized error handling and logging.
Idempotent	An operation is idempotent if performing it multiple times has the same effect as performing it once. For example, issuing the same `CREATE` API call twice should result in one resource, not two. A core requirement for **provider** operations.	Ensures safety of retries and re-running plans.
Jitter	Random variation added to retry delays (e.g., +/- 0.5 seconds). Prevents many clients from retrying simultaneously (a "thundering herd" problem) after a service outage, which could cause a new outage upon recovery.	Used with **exponential backoff**.
`JsonSerializable`	A mixin class providing standard `to_dict`, `from_dict`, `to_json`, and `from_json` methods. Core data types like `Resource` and `StateRecord` inherit from it to simplify serialization for the **state file**.	
`LockHandle`	An object returned by `acquire_lock` representing an acquired state lock. It contains metadata (lock path, process ID, lock ID) and manages a **heartbeat thread** to prevent the lock from becoming **stale**.	Must be passed to `release_lock` to cleanly release the lock.
`PartialApplyError`	A special exception raised when an `apply` operation completes for some resources but fails for others. It contains a dictionary of succeeded resources so the state can be partially updated.	Prevents losing track of **zombie resources** created before the failure.
`PermanentProviderError`	A subclass of `ProviderError` indicating a failure that is not retryable (e.g., invalid credentials, unsupported resource type, validation error). The engine should stop immediately.	Contrast with `TransientProviderError`.
`PlanAction`	A data structure representing a single change to be made to a resource. It includes the `action_type` (`CREATE`, `UPDATE`, `DELETE`, `NOOP`), the `resource` (desired state), and optionally the `prior_state` and `new_state`.	The building block of an **execution plan**. Generated by `generate_plan`.

Term	Definition	Related Concepts & Notes
Planner	The component (**Component Design: Planner**) responsible for building the **dependency graph**, performing **topological sort**, and generating the **execution plan**. It answers "what needs to change and in what order?"	Key methods: `build_graph`, `validate_acyclic`, `topological_sort`, `generate_plan`.
Provider	A plugin that translates the engine's generic CRUD operations into API calls for a specific cloud platform or service (e.g., AWS, Azure, Google Cloud). Implements the `BaseProvider` interface.	The extensibility point of the system; new clouds are supported by writing new providers.
`ProviderConfig`	A data structure holding configuration for a specific provider instance, such as credentials, region, and endpoint overrides.	Fields: `provider_type`, `config`.
`ProviderError`	The base exception class for errors originating from a **provider** (e.g., API failures). Subclassed into `TransientProviderError` and `PermanentProviderError`.	
Rate Limiting	Restrictions imposed by cloud APIs on the frequency or volume of requests from a client. Exceeding limits results in throttling errors (HTTP 429).	Must be handled by providers using **exponential backoff** and **jitter**.
`Resource`	The fundamental data structure representing a unit of infrastructure as defined in configuration (e.g., an AWS EC2 instance, a Google Cloud Storage bucket). It has a type, a unique name within that type, and a set of attributes.	Fields: `id`, `type`, `name`, `attributes`. The `id` is initially empty and populated by the provider after creation.
State Drift	The situation where the actual state of a resource in the cloud diverges from the state recorded in the **state file**. This can happen due to manual changes, direct API calls, or other IaC tools.	The engine's `refresh_state` operation can detect drift by calling `provider.read`.
State File	A persistent file (typically JSON) that stores the last known state of all managed infrastructure as a dictionary of `StateRecord` objects, keyed by resource address. It is the **single source of truth** for what the engine believes is deployed.	Managed by the **State Manager**.
State Inspection	The process of analyzing the **state file** for issues like orphaned resources, duplicates, or validation errors. Performed by the `StateInspector` class and the `inspect_state_command` CLI command.	Useful for debugging and maintenance.
`StateInspector`	A utility class providing methods to load, analyze, and compare state files.	Methods: `load_state`, `analyze`, `compare`.
State Manager	The component (**Component Design: State Manager**) responsible for persisting and managing the **state file**, handling **locking** to prevent concurrent modifications, and computing **diffs** between states.	Key methods: `read_state`, `write_state`, `acquire_lock`, `compute_diff`.
`StateRecord`	A data structure representing a snapshot of a `Resource` as it exists (or is believed to exist) in the cloud. It is what is stored in the **state file**. It includes the cloud-assigned `resource_id` and tracks `dependencies`.	Fields: `resource_id`, `resource_type`, `resource_name`, `attributes`, `dependencies`.
`StateReport`	The output of a state inspection, containing statistics and identified issues.	Fields: `total_resources`, `resources_by_type`, `orphaned_resources`, `duplicate_addresses`, `validation_errors`.
Structured Logging	A logging practice where log messages are emitted in a machine-parsable format (like JSON) with consistent, named fields (e.g., `timestamp`, `level`, `correlation_id`, `component`, `message`). Greatly aids in debugging and log analysis.	Configured via `setup_logging`.
Topological Sort	A linear ordering of the nodes in a **DAG** such that for every directed edge from node *u* to node *v*, *u* appears before *v* in the ordering. This provides a safe execution order for resources.	Implemented by the `topological_sort` function in the **Planner**.

Term	Definition	Related Concepts & Notes
`TransientProviderError`	A subclass of `ProviderError` indicating a failure that might be temporary and is worth retrying (e.g., network timeout, rate limit, internal server error). The engine will apply **retry logic**.	Contrast with `PermanentProviderError`.
Zombie Resource	A resource that exists in the cloud but is not recorded in the **state file**. This can happen if a resource is created (e.g., by a **partial apply** failure) but the state update fails. The engine loses the ability to manage it.	Mitigated by **atomic state updates** and careful error recovery in the **Executor**.