```
// Core instruction model (already parsed from DSL)

export type Facet = string;
export type SpellName = string;
export type ClothName = string;
export type FusionName = string;

export interface Origin {
  facets: Facet[];
  system: SystemDef;
}

export interface SystemDef {
  name: string;
  layers: LayerDef[];
  finalize: string;
}

export interface LayerDef {
  name: string;
  elements: ElementDef[];
}

export type ElementDef =
  | WrapDef
  | ChainDef
  | NestDef
  | EmergeDef;

export interface WrapDef {
  kind: "wrap";
  target: SpellName | ClothName;
}

export interface ChainDef {
  kind: "chain";
  target: SpellName | ClothName;
}

export interface NestDef {
  kind: "nest";
  name: string;
  elements: ElementDef[];
}
```

```
export interface EmergeDef {
  kind: "emerge";
  fusion: FusionName;
  elements: ElementDef[];
}

// Runtime module contracts

export interface CodexModuleContext {
  system: SystemState;
  layer: LayerState;
  env: Record<string, unknown>;
}

export interface CodexModule {
  name: string;
  apply(ctx: CodexModuleContext): Promise<void> | void;
}

export interface CodexFusion {
  name: string;
  apply(ctx: CodexModuleContext, elements: ElementDef[]): Promise<void> | void;
}

// Runtime state

export interface SystemState {
  name: string;
  facets: Facet[];
  data: Record<string, unknown>;
}

export interface LayerState {
  name: string;
  data: Record<string, unknown>;
}

// Registry

export interface CodexRegistry {
  spells: Map<SpellName, CodexModule>;
  cloths: Map<ClothName, CodexModule>;
  fusions: Map<FusionName, CodexFusion>;
```

```
}

export function createRegistry(): CodexRegistry {
  return {
    spells: new Map(),
    cloths: new Map(),
    fusions: new Map(),
  };
}

export function registerSpell(reg: CodexRegistry, mod: CodexModule) {
  reg.spells.set(mod.name, mod);
}

export function registerCloth(reg: CodexRegistry, mod: CodexModule) {
  reg.cloths.set(mod.name, mod);
}

export function registerFusion(reg: CodexRegistry, fusion: CodexFusion) {
  reg.fusions.set(fusion.name, fusion);
}

// VM execution

export interface CodexVMOptions {
  env?: Record<string, unknown>;
}

export class CodexVM {
  private readonly origin: Origin;
  private readonly registry: CodexRegistry;
  private readonly env: Record<string, unknown>;

  constructor(origin: Origin, registry: CodexRegistry, options: CodexVMOptions = {}) {
    this.origin = origin;
    this.registry = registry;
    this.env = options.env ?? {};
  }

  async run(): Promise<SystemState> {
    const systemState: SystemState = {
      name: this.origin.system.name,
      facets: [...this.origin.facets],
      data: {},
```

```
  };

  for (const layerDef of this.origin.system.layers) {
    const layerState: LayerState = { name: layerDef.name, data: {} };
    await this.executeLayer(systemState, layerState, layerDef);
  }

  // FINALIZE is a semantic marker; VM returns final system state
  return systemState;
}

private async executeLayer(
  system: SystemState,
  layer: LayerState,
  layerDef: LayerDef
): Promise<void> {
  for (const el of layerDef.elements) {
    await this.executeElement(system, layer, el);
  }
}

private async executeElement(
  system: SystemState,
  layer: LayerState,
  el: ElementDef
): Promise<void> {
  switch (el.kind) {
    case "wrap":
      await this.invokeModule(el.target, system, layer);
      break;
    case "chain":
      await this.invokeModule(el.target, system, layer);
      break;
    case "nest":
      await this.executeNested(system, layer, el);
      break;
    case "emerge":
      await this.executeFusion(system, layer, el);
      break;
  }
}

private async invokeModule(
  name: SpellName | ClothName,
```

```
    system: SystemState,
    layer: LayerState
  ): Promise<void> {
    const mod =
      this.registry.spells.get(name) ??
      this.registry.cloths.get(name);

    if (!mod) {
      throw new Error(`Module not found: ${name}`);
    }

    const ctx: CodexModuleContext = { system, layer, env: this.env };
    await mod.apply(ctx);
  }

  private async executeNested(
    system: SystemState,
    parentLayer: LayerState,
    nest: NestDef
  ): Promise<void> {
    const nestedLayer: LayerState = {
      name: `${parentLayer.name}.${nest.name}`,
      data: {},
    };

    for (const el of nest.elements) {
      await this.executeElement(system, nestedLayer, el);
    }
  }

  private async executeFusion(
    system: SystemState,
    layer: LayerState,
    emerge: EmergeDef
  ): Promise<void> {
    const fusion = this.registry.fusions.get(emerge.fusion);
    if (!fusion) {
      throw new Error(`Fusion not found: ${emerge.fusion}`);
    }

    const ctx: CodexModuleContext = { system, layer, env: this.env };
    await fusion.apply(ctx, emerge.elements);
  }
}
```