

AgentApiClientTests.cs

```
using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Security.Cryptography;
using BridgeControl.Agent;

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Verifies agent API client behavior and regression scenarios.
/// </summary>
public sealed class AgentApiClientTests : IDisposable
{
    private readonly string directory = Path.Combine(Path.GetTempPath(),
"BridgeControlAgentHttpTests", Guid.NewGuid().ToString("N"));
    private readonly DesiredState desired = new(Guid.NewGuid(), Guid.NewGuid(),
Guid.NewGuid(), DateTimeOffset.UtcNow);
    private static readonly byte[] Artifact = $"{\"port\":8080}\"u8.ToArray();
    /// <summary>
    /// Initializes a new <see cref="AgentApiClientTests"/> instance and its dependencies.
    /// </summary>
    public AgentApiClientTests() => Directory.CreateDirectory(directory);
    /// <summary>
    /// Creates an agent API client backed by a controlled HTTP handler.
    /// </summary>
    private AgentApiClient Client(HttpMessageHandler handler) => new(new HttpClient(handler),
new AgentOptions
    {
        ApiBaseUrl = "https://bridge.example/", DataDirectory = directory
    }, () => "private-machine-credential");
    /// <summary>
    /// Builds an HTTP artifact response with the supplied bytes and metadata.
    /// </summary>
    private HttpResponseMessage ArtifactResponse()
    {
        HttpResponseMessage response = new(HttpStatusCode.OK) { Content = new
ByteArrayContent(Artifact) };
        byte[] hash = SHA256.HashData(Artifact);
        response.Headers.ETag = new('"' + Convert.ToHexStringLower(hash) + '"');
        response.Headers.Add("Content-Digest",
$"sha-256={Convert.ToBase64String(hash)}");
        response.Headers.Add("X-Product-Version-Id", desired.ProductVersionId.ToString());
        response.Headers.Add("X-Effective-Configuration-Revision-Id",
desired.EffectiveConfigurationRevisionId.ToString());
        return response;
    }
    /// <summary>
    /// Verifies that assignment discovery reads all pages including empty intermediate page.
    /// </summary>
    [Fact]
    public async Task AssignmentDiscovery_ReadsAllPagesIncludingEmptyIntermediatePage()
    {
        Guid id = Guid.NewGuid();
        var handler = new Handler(request => new(HttpStatusCode.OK)
        {
            Content = JsonContent.Create(request.RequestUri!.Query.Contains("page=1&")
? new AssignmentPage([], 1, 100, 101)
```

```

        : new AssignmentPage([new(id, Guid.NewGuid(), Guid.NewGuid(),
null)], 2, 100, 101))
    });
    var rows = await Client(handler).AssignmentsAsync(default);
    Assert.Equal(id, Assert.Single(rows).Id);
    Assert.Equal(2, handler.Addresses.Count(x =>
x.AbsolutePath.EndsWith("/installations")));
}
/// <summary>
/// Verifies that verified download promotes only complete artifact.
/// </summary>
[Fact]
public async Task VerifiedDownload_PromotesOnlyCompleteArtifact()
{
    AgentApiClient client = Client(new Handler(_ => ArtifactResponse()));
    string path = Path.Combine(directory, "config.json");
    Download result = await
client.DownloadAsync("/api/v1/agent/artifacts/test/content", desired, path, default);
    Assert.Equal(Artifact, File.ReadAllBytes(path));
    Assert.Equal(Convert.ToHexStringLower(SHA256.HashData(Artifact)), result.Hash);
    Assert.False(File.Exists(path + ".partial"));
}
/// <summary>
/// Verifies that invalid artifact does not replace previous file.
/// </summary>
[Theory]
[InlineData("length")]
[InlineData("hash")]
[InlineData("revision")]
[InlineData("version")]
[InlineData("digest")]
[InlineData("weak")]
[InlineData("partial")]
public async Task InvalidArtifact_DoesNotReplacePreviousFile(string corruption)
{
    AgentApiClient client = Client(new Handler(_ =>
    {
        HttpResponseMessage response = ArtifactResponse();
        switch (corruption)
        {
            case "length": response.Content.Headers.ContentLength =
Artifact.Length + 2; break;
            case "hash": response.Content = new ByteArrayContent(new
byte[Artifact.Length]); break;
            case "revision":
response.Headers.Remove("X-Effective-Configuration-Revision-Id");
response.Headers.Add("X-Effective-Configuration-Revision-Id", Guid.NewGuid().ToString()); break;
            case "version": response.Headers.Remove("X-Product-Version-Id");
response.Headers.Add("X-Product-Version-Id", Guid.NewGuid().ToString()); break;
            case "digest": response.Headers.Remove("Content-Digest"); break;
            case "weak": response.Headers.ETag =
new(response.Headers.ETag!.Tag, true); break;
            case "partial": response.StatusCode =
HttpStatusCode.PartialContent; break;
        }
        return response;
    }));
    string path = Path.Combine(directory, "config.json");
    File.WriteAllText(path, "previous");
    await Assert.ThrowsAsync<InvalidDataException>(() =>
client.DownloadAsync("/api/v1/agent/artifacts/test/content", desired, path, default));
    Assert.Equal("previous", File.ReadAllText(path));
}

```

```

        Assert.False(File.Exists(path + ".partial"));
    }
    /// <summary>
    /// Verifies that foreign link is rejected before sending credentials.
    /// </summary>
    [Fact]
    public async Task ForeignLink_IsRejectedBeforeSendingCredentials()
    {
        Handler handler = new(_ => new HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = JsonContent.Create(desired),
            Headers = { { "Link",
"<https://attacker.example/api/v1/agent/artifacts/test/content>; rel=\"installer\"" } }
        });
        await Assert.ThrowsAsync<InvalidDataException>(() =>
Client(handler).DesiredAsync(desired.InstallationId, default));
        Assert.All(handler.Addresses, address => Assert.Equal("bridge.example",
address.Host));
    }
    /// <summary>
    /// Verifies that pagination follows next even when eligible page is empty.
    /// </summary>
    [Fact]
    public async Task Pagination_FollowsNextEvenWhenEligiblePageIsEmpty()
    {
        Guid deploymentId = Guid.NewGuid();
        Handler handler = new(request => request.RequestUri!.Query.Contains("page=1")
? new HttpResponseMessage(HttpStatusCode.OK)
{
    Content = JsonContent.Create(Array.Empty<Deployment>()),
    Headers = { { "Link",
"</api/v1/agent/deployments?status=pending&page=2&pageSize=100>; rel=\"next\"" } }
}
: new(HttpStatusCode.OK) { Content = JsonContent.Create(new[] { new
Deployment(deploymentId, desired.InstallationId, desired.ProductVersionId,
desired.EffectiveConfigurationRevisionId, "Install") }) });
        IReadOnlyList<Deployment> result = await Client(handler).PendingAsync(default);
        Assert.Equal(deploymentId, Assert.Single(result).Id);
    }
    /// <summary>
    /// Verifies that unauthorized refreshes token once and replays same request.
    /// </summary>
    [Fact]
    public async Task Unauthorized_RefreshesTokenOnceAndReplaysSameRequest()
    {
        int calls = 0;
        Handler handler = new(_ => new(++calls == 1 ? HttpStatusCode.Unauthorized :
HttpStatusCode.NoContent));
        await Client(handler).HeartbeatAsync(new("0.1.0", "Online",
DateTimeOffset.UtcNow), default);
        Assert.Equal(2, handler.TokenRequests);
        Assert.Equal(2, calls);
        Assert.Equal(handler.Bodies[0], handler.Bodies[1]);
    }
    /// <summary>
    /// Verifies that repeated unauthorized stops after one refresh.
    /// </summary>
    [Fact]
    public async Task RepeatedUnauthorized_StopsAfterOneRefresh()
    {
        Handler handler = new(_ => new(HttpStatusCode.Unauthorized));
    }

```

```

        AgentApiException error = await Assert.ThrowsAsync<AgentApiException>(() =>
Client(handler).HeartbeatAsync(new("0.1.0", "Online", DateTimeOffset.UtcNow), default));
        Assert.Equal(HttpStatusCode.Unauthorized, error.Status);
        Assert.Equal(2, handler.TokenRequests);
    }
    /// <summary>
    /// Verifies that rate limit honors retry after before next request.
    /// </summary>
    [Fact]
    public async Task RateLimit_HonorsRetryAfterBeforeNextRequest()
    {
        int calls = 0;
        Handler handler = new(_ =>
        {
            calls++;
            HttpResponseMessage response = new(HttpStatusCode.TooManyRequests);
            response.Headers.RetryAfter = new
RetryConditionHeaderValue(TimeSpan.FromMinutes(1));
            return response;
        });
        AgentApiClient client = Client(handler);
        await Assert.ThrowsAsync<AgentApiException>(() =>
client.HeartbeatAsync(new("0.1.0", "Online", DateTimeOffset.UtcNow), default));
        using CancellationTokenSource cancel = new(100);
        await Assert.ThrowsAnyAsync<OperationCanceledException>(() =>
client.HeartbeatAsync(new("0.1.0", "Online", DateTimeOffset.UtcNow), cancel.Token));
        Assert.Equal(1, calls);
    }
    /// <summary>
    /// Verifies that invalid origins are rejected.
    /// </summary>
    [Theory]
    [InlineData("http://bridge.example/")]
    [InlineData("https://user:secret@bridge.example/")]
    [InlineData("https://bridge.example/api/")]
    public void InvalidOrigins_AreRejected(string url)
    {
        Assert.Throws<InvalidOperationException>(() => new AgentOptions { ApiBaseUrl = url
}.Validate());
    }
    /// <summary>
    /// Verifies that protected state round trips without plaintext and rejects corruption.
    /// </summary>
    [Fact]
    public void ProtectedState_RoundTripsWithoutPlaintextAndRejectsCorruption()
    {
        ProtectedStore store = new(directory);
        store.Save(new() { Identity = new(Guid.NewGuid(), "private-test-credential",
DateTimeOffset.UtcNow.AddDays(1)) });
        Assert.Equal("private-test-credential", store.Load().Identity!.Credential);
        string path = Path.Combine(directory, "state.dat");
        Assert.DoesNotContain("private-test-credential",
System.Text.Encoding.UTF8.GetString(File.ReadAllBytes(path)));
        byte[] damaged = File.ReadAllBytes(path);
        damaged[damaged.Length / 2] ^= 127;
        File.WriteAllBytes(path, damaged);
        Assert.Throws<CryptographicException>(() => store.Load());
    }
    /// <summary>
    /// Releases the resources owned by this instance.
    /// </summary>
    public void Dispose() { TestDirectory.Delete(directory, "BridgeControlAgentHttpTests"); }

```

```

    /// <summary>
    /// Routes test HTTP requests to a supplied response callback.
    /// </summary>
    private sealed class Handler(Func<HttpRequestMessage, HttpResponseMessage> respond) :
    HttpResponseMessageHandler
    {
        public int TokenRequests { get; set; }
        public List<Uri> Addresses { get; } = [];
        public List<string?> Bodies { get; } = [];
        /// <summary>
        /// Sends the test HTTP request with the supplied method, credentials, and
payload.
        /// </summary>
        protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request, CancellationToken cancellationToken)
        {
            Addresses.Add(request.RequestUri!);
            if (request.RequestUri!.AbsolutePath.EndsWith("/auth/token"))
            {
                TokenRequests++;
                Assert.Null(request.Headers.Authorization);
                return new(HttpStatusCode.OK) { Content = JsonContent.Create(new
Token("access-" + TokenRequests, DateTimeOffset.UtcNow.AddMinutes(10))) };
            }
            Assert.Equal("Bearer", request.Headers.Authorization?.Scheme);
            Bodies.Add(request.Content is null ? null : await
request.Content.ReadAsStringAsync(cancellationToken));
            return respond(request);
        }
    }
}

```

AgentServerMigrationTests.cs

```

using System.Net;
using System.Net.Http.Json;
using System.Text.Json;
using BridgeControl.Agent;

namespace BridgeControl.Agent.Tests;

public sealed class AgentServerMigrationTests
{
    private static AgentOptions Options(string url = "https://new.example:7443/") => new() {
    ApiBaseUrl = url };
    private static AgentState Enrolled() => new()
    {
        ApiBaseUrl = "https://old.example/",
        Identity = new(Guid.NewGuid(), "existing-credential",
DateTimeOffset.UtcNow.AddDays(1)),
        LastReportAt = DateTimeOffset.UtcNow,
        Installations = [new() { Id = Guid.NewGuid(), ProductId = Guid.NewGuid(),
RegistrationName = "existing" }],
        Execution = new()
        {
            Deployment = new(Guid.NewGuid(), Guid.NewGuid(), Guid.NewGuid(), null,
"Upgrade"),
            Target = new(new(Guid.NewGuid(), Guid.NewGuid(), Guid.NewGuid(),
DateTimeOffset.UtcNow), "/api/v1/agent/artifacts/msi/content",
"/api/v1/agent/artifacts/config/content"),

```

```

        Phase = "Interrupted"
    }
};
private static HttpResponseMessage TokenResponse(Guid id, string identityType = "agent",
bool expired = false)
{
    string payload = Convert.ToBase64String(JsonSerializer.SerializeToUtf8Bytes(new {
sub = id, identity_type = identityType }));
    .TrimEnd('=').Replace('+', '-').Replace('/', '_');
    return new(HttpStatusCode.OK)
    {
        Content = JsonContent.Create(new Token($"header.{payload}.signature",
DateTimeOffset.UtcNow.AddMinutes(expired ? -10 : 10)))
    };
}

[Fact]
public async Task VerifiedMove_ChangesOnlyOrigin_AndCanBeRetried()
{
    Store store = new(Enrolled());
    string original = JsonSerializer.Serialize(store.State);
    using HttpClient http = new(new Handler(async request =>
    {
        Assert.Equal("https://new.example:7443/api/v1/agent/auth/token",
request.RequestUri!.AbsoluteUri);
        Assert.Equal(HttpMethod.Post, request.Method);
        using JsonDocument body = JsonDocument.Parse(await
request.Content!.ReadAsStringAsync());
        Assert.Equal(store.State.Identity!.Credential,
body.RootElement.GetProperty("credential").GetString());
        return TokenResponse(store.State.Identity.AgentId);
    }));
    await AgentServerMigration.ChangeAsync(Options(), store, http, default);
    await AgentServerMigration.ChangeAsync(Options(), store, http, default);
    Assert.Equal(2, store.Saves);
    Assert.Equal("https://new.example:7443/", store.State.ApiBaseUrl);
    Assert.Equal(original.Replace("https://old.example/",
"https://new.example:7443/"), JsonSerializer.Serialize(store.State));
}

[Theory]
[InlineData("unauthorized")]
[InlineData("redirect")]
[InlineData("wrong-agent")]
[InlineData("wrong-type")]
[InlineData("expired")]
[InlineData("malformed")]
[InlineData("unreachable")]
[InlineData("cancelled")]
public async Task FailedVerification_DoesNotChangeEnrollment(string failure)
{
    Store store = new(Enrolled());
    string original = JsonSerializer.Serialize(store.State);
    using HttpClient http = new(new Handler(_ => Task.FromResult(failure switch
    {
        "unauthorized" => new HttpResponseMessage(HttpStatusCode.Unauthorized),
        "redirect" => new HttpResponseMessage(HttpStatusCode.TemporaryRedirect) {
Headers = { Location = new("https://elsewhere.example/") } },
        "wrong-agent" => TokenResponse(Guid.NewGuid()),
        "wrong-type" => TokenResponse(store.State.Identity!.AgentId, "human"),
        "expired" => TokenResponse(store.State.Identity!.AgentId, expired: true),
    }));

```

```

        "malformed" => new HttpResponseMessage(HttpStatusCode.OK) { Content =
JsonContent.Create(new Token("invalid", DateTimeOffset.UtcNow.AddMinutes(10))) },
        "unreachable" => throw new HttpRequestException(),
        _ => throw new OperationCanceledException()
    }));
    await Assert.ThrowsAnyAsync<Exception>(() =>
AgentServerMigration.ChangeAsync(Options(), store, http, default));
    Assert.Equal(0, store.Saves);
    Assert.Equal(original, JsonSerializer.Serialize(store.State));
}

[Theory]
[InlineData("unenrolled")]
[InlineData("pending")]
[InlineData("missing-origin")]
[InlineData("http")]
public async Task InvalidMigration_SendsNoCredentials(string failure)
{
    Store store = new(Enrolled());
    if (failure == "unenrolled") { store.State.Identity = null; }
    if (failure == "pending") { store.State.EnrollmentPending = true; }
    if (failure == "missing-origin") { store.State.ApiBaseUrl = null; }
    int requests = 0;
    using HttpClient http = new(new Handler(_ => { requests++; return
Task.FromResult(new HttpResponseMessage(HttpStatusCode.OK)); }));
    await Assert.ThrowsAsync<InvalidOperationException>(() =>
AgentServerMigration.ChangeAsync(
        Options(failure == "http" ? "http://new.example/" :
"https://new.example/"), store, http, default));
    Assert.Equal(0, requests);
    Assert.Equal(0, store.Saves);
}

[Fact]
public async Task SaveFailure_RestoresInMemoryOrigin()
{
    Store store = new(Enrolled()) { FailSave = true };
    string original = JsonSerializer.Serialize(store.State);
    using HttpClient http = new(new Handler(_ =>
Task.FromResult(TokenResponse(store.State.Identity!.AgentId))));
    await Assert.ThrowsAsync<IOException>(() =>
AgentServerMigration.ChangeAsync(Options(), store, http, default));
    Assert.Equal(original, JsonSerializer.Serialize(store.State));
}

private sealed class Store(AgentState state) : IStateStore
{
    public AgentState State { get; private set; } = state;
    public int Saves { get; private set; }
    public bool FailSave { get; init; }
    public AgentState Load() => State;
    public void Save(AgentState state)
    {
        if (FailSave) { throw new IOException(); }
        Saves++;
        State = state;
    }
}

private sealed class Handler(Func<HttpRequestMessage, Task<HttpResponseMessage>> respond)
: HttpMessageHandler
{

```

```

        protected override Task<HttpResponseBody> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken) => respond(request);
    }
}

```

ApiIntegrationTests.cs

```

using System.Net;
using BridgeControl.Agent;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging.Abstractions;

namespace BridgeControl.Agent.Tests;

// Reuse the API's disposable SQL/server fixture; the production agent has no backend dependency.
/// <summary>
/// Verifies api integration behavior and regression scenarios.
/// </summary>
public sealed class ApiIntegrationTests : EffectiveConfigurationTest
{
    /// <summary>
    /// Verifies that agent completes enrollment download deployment and reconciliation
    against real api.
    /// </summary>
    [Theory]
    [InlineData("Service")]
    [InlineData("Desktop")]
    public async Task
AgentCompletesEnrollmentDownloadDeploymentAndReconciliationAgainstRealApi(string kind)
    {
        await PrepareAsync();
        (Guid product, Guid version) = await CatalogAsync(installer: true);
        Guid customer = await CustomerAsync();
        using HttpResponseMessage enrollment = await SendAsync(HttpMethod.Post,
"/api/v1/agent-enrollments", new { CustomerId = customer });
        Assert.Equal(HttpStatusCode.Created, enrollment.StatusCode);
        string code = (await
ReadAsync(enrollment)).GetProperty("enrollmentCode").GetString()!;
        ApplicationOptions application = new()
        {
            Kind = kind,
            ProductId = product, Name = "MVP integration", UpgradeCode =
Guid.NewGuid(), ServiceName = "FixtureOnly",
            ConfigurationPath = Path.Combine(Factory.PrivateRoot, "application",
"config.json"), Versions = [new(version, Guid.NewGuid(), "1.0.0")]
        };
        AgentOptions options = new()
        {
            ApiBaseUrl = "https://localhost/", DataDirectory =
Path.Combine(Factory.PrivateRoot, "agent"), Applications = []
        };
        Directory.CreateDirectory(options.DataDirectory);
        using HttpClient transport = Factory.CreateApiClient();
        Registration? identity = null;
        AgentApiClient api = new(transport, options, () => identity!.Credential);
        identity = await api.RegisterAsync(code, default);
    }
}

```

```

        using var settings = await SendAsync(HttpMethod.Put,
$"/api/v1/products/{product}/versions/{version}/deployment-settings",
        new { Kind = kind, ConfigurationPath = application.ConfigurationPath,
ExecutablePath = kind == "Desktop" ? Path.Combine(Factory.PrivateRoot, "application",
"widget.exe") : null, ServiceName = kind == "Service" ? "FixtureOnly" : null });
        Assert.Equal(HttpStatusCode.OK, settings.StatusCode);
        using HttpResponseMessage published = await PublishAsync(product, version);
        Assert.Equal(HttpStatusCode.OK, published.StatusCode);
        using var assignment = await SendAsync(HttpMethod.Post, "/api/v1/installations",
new { AgentId = identity.AgentId, ProductId = product, Name = "Assigned application" });
        Assert.Equal(HttpStatusCode.Created, assignment.StatusCode);
        await api.HeartbeatAsync(new (AgentRuntime.Version, "Online", api.UtcNow),
default);
        DeploymentEngineTests.MemoryStore store = new() { Value = new() { Identity =
identity } };
        DeploymentEngineTests.FakeMachine machine = new() { TargetVersion = version };
        DeploymentEngine Engine() => new(options, api, store, machine,
NullLogger<DeploymentEngine>.Instance);
        await Engine().CycleAsync(default);
        Guid installation = Assert.Single(store.Value.Installations).Id;
        Guid configuration = await GenerateAsync(installation, version);
        using HttpResponseMessage license = await SendAsync(HttpMethod.Post,
"/api/v1/licenses", new
        {
            CustomerId = customer, ProductId = product, LicenseReference = "Agent MVP
test", ValidFrom = DateTimeOffset.UtcNow.AddDays(-1), MaxInstallations = 1
        });
        Assert.Equal(HttpStatusCode.Created, license.StatusCode);
        using HttpResponseMessage assigned = await SendAsync(HttpMethod.Put,
$"/api/v1/installations/{installation}/license", new { LicenseId = (await
ReadAsync(license)).GetProperty("id").GetGuid() });
        Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode);
        using HttpResponseMessage desired = await SendAsync(HttpMethod.Put,
$"/api/v1/installations/{installation}/desired-state", new { ProductVersionId = version,
EffectiveConfigurationRevisionId = configuration });
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        await ReconcileAsync();
        await Engine().CycleAsync(default);
        Assert.Equal(1, machine.InstallCalls);
        Assert.Equal(1, machine.ConfigureCalls);
        Assert.Null(store.Value.Execution);
        await Engine().CycleAsync(default);
        await ReconcileAsync();
        await using BridgeControl.Data.VendorDbContext db = await
OpenVendorAsync(VendorId);
        BridgeControl.Models.Vendor.Deployment deployment = await
db.Deployments.SingleAsync(d => d.InstallationId == installation);
        Assert.Equal("Reconciled", deployment.Status);
        Assert.Equal("Succeeded", (await db.DeploymentAttempts.SingleAsync(a =>
a.DeploymentId == deployment.Id)).Status);
        Assert.Equal(configuration, (await db.InstallationStateReports.OrderByDescending(r
=> r.ReportedAt).FirstAsync(r => r.InstallationId ==
installation)).EffectiveConfigurationRevisionId);
        Assert.Equal(identity.AgentId, store.Load().Identity!.AgentId);
        Assert.Equal(1, machine.InstallCalls);
    }
    /// <summary>
    /// Processes pending installation-state reconciliation for the test vendor.
    /// </summary>
    private async Task ReconcileAsync()
    {
        await using AsyncServiceScope scope = Factory.Services.CreateAsyncScope();

```

```

        await
scope.ServiceProvider.GetRequiredService<StateReconciliationProcessor>().ProcessVendorAsync (VendorId);
    }
}

```

BridgeControl.Agent.Tests.csproj

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net10.0-windows</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
    <IsTestProject>true</IsTestProject>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.14.1" />
    <PackageReference Include="xunit" Version="2.9.3" />
    <PackageReference Include="xunit.runner.visualstudio" Version="3.1.4" PrivateAssets="all" />
    <ProjectReference Include="../BridgeControl.Agent/BridgeControl.Agent.csproj" />
    <ProjectReference Include="../BridgeControl.Tests/BridgeControl.Tests.csproj" />
    <Using Include="Xunit" />
  </ItemGroup>
</Project>

```

DeploymentEngineTests.cs

```

using System.Net;
using System.Security.Cryptography;
using System.Text.Json;
using BridgeControl.Agent;
using Microsoft.Extensions.Logging.Abstractions;

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Verifies deployment engine behavior and regression scenarios.
/// </summary>
public sealed class DeploymentEngineTests : IDisposable
{
    private readonly string directory = Path.Combine(Path.GetTempPath(),
"BridgeControlAgentTests", Guid.NewGuid().ToString("N"));
    private readonly MemoryStore store = new();
    private readonly FakeApi api = new();
    private readonly FakeMachine machine = new();
    private readonly AgentOptions options;
    /// <summary>
    /// Initializes a new <see cref="DeploymentEngineTests"/> instance and its dependencies.
    /// </summary>
    public DeploymentEngineTests()
    {
        Directory.CreateDirectory(directory);
        ApplicationOptions app = new()
        {
            ProductId = Guid.NewGuid(), Name = "Test service", UpgradeCode =
Guid.NewGuid(), ServiceName = "TestService",

```

```

        ConfigurationPath = Path.Combine(directory, "app.json"), Versions =
[new(api.State.ProductVersionId, Guid.NewGuid(), "1.0.0")]
    };
    options = new() { ApiBaseUrl = "https://bridge.example/", DataDirectory =
Path.Combine(directory, "agent"), Applications = [app] };
    store.Value = new()
    {
        Identity = new(api.AgentId, "secret", DateTimeOffset.UtcNow.AddDays(90)),
        Installations = [new() { Id = api.State.InstallationId, ProductId =
app.ProductId, RegistrationName = app.Name }]
    };
    machine.TargetVersion = api.State.ProductVersionId;
}
/// <summary>
/// Creates a deployment engine using the test API, state store, and machine.
/// </summary>
private DeploymentEngine Engine() => new(options, api, store, machine,
NullLogger<DeploymentEngine>.Instance);
/// <summary>Duplicate historical MSI mappings do not block reporting and upgrading to a
unique new MSI.</summary>
[Fact]
public async Task DuplicateHistoricalMsi_ReportsAppliedRelease_AndUpgradesToNewIdentity()
{
    ApplicationOptions app = options.Applications[0];
    Guid oldVersion = Guid.NewGuid(), oldCode = Guid.NewGuid();
    app.Versions[0] = app.Versions[0] with { Version = "1.0.2" };
    app.Versions.Add(new(oldVersion, oldCode, "1.0.0"));
    app.Versions.Add(new(Guid.NewGuid(), oldCode, "1.0.0"));
    app.SettingsVersionId = api.State.ProductVersionId;
    options.Applications = [];
    api.Assignments = [new(api.State.InstallationId, app.ProductId, api.AgentId,
app)];

    api.Operation = "Update";
    machine.InstalledVersion = oldVersion;
    machine.ResolveMsiIdentity = true;
    store.Value.Installations[0].Applied = new(oldVersion, Guid.NewGuid(),
FakeApi.Hash, null);
    await Engine().CycleAsync(default);
    Assert.Equal(oldVersion, api.Reports[0].InstalledProductVersionId);
    Assert.Equal("Succeeded", api.Terminal?.Status);
    Assert.Equal(1, machine.InstallCalls);
    await Engine().CycleAsync(default);
    Assert.Equal(api.State.ProductVersionId,
api.Reports.Last().InstalledProductVersionId);
    Assert.Equal(api.State.EffectiveConfigurationRevisionId,
api.Reports.Last().EffectiveConfigurationRevisionId);
    Assert.Equal(1, machine.InstallCalls);
}
/// <summary>
/// Verifies that api assignment deploys with no local applications and keeps stable
identity.
/// </summary>
[Fact]
public async Task ApiAssignment_DeploysWithNoLocalApplications_AndKeepsStableIdentity()
{
    var app = options.Applications.Single();
    options.Applications = [];
    store.Value.Installations.Clear();
    api.Assignments = [new(api.State.InstallationId, app.ProductId, api.AgentId,
app)];

    await Engine().CycleAsync(default);
    Assert.Equal("Succeeded", api.Terminal!.Status);

```

```

        Assert.Equal(api.State.InstallationId,
Assert.Single(store.Value.Installations).Id);
        Assert.Equal(0, api.RegistrationCalls);
        await Engine().CycleAsync(default);
        Assert.Single(store.Value.Installations);
        Assert.Equal(1, machine.InstallCalls);
    }
    /// <summary>
    /// Verifies that desired version changed since discovery waits for fresh application
settings.
    /// </summary>
    [Fact]
    public async Task DesiredVersionChangedSinceDiscovery_WaitsForFreshApplicationSettings()
    {
        options.Applications[0].SettingsVersionId = Guid.NewGuid();
        await Engine().CycleAsync(default);
        Assert.Equal(0, machine.InstallCalls);
        Assert.Null(store.Value.Execution);
    }
    /// <summary>
    /// Verifies that foreign assignment cannot create local state or execute.
    /// </summary>
    [Fact]
    public async Task ForeignAssignment_CannotCreateLocalStateOrExecute()
    {
        api.Assignments = [new(Guid.NewGuid(), options.Applications[0].ProductId,
Guid.NewGuid(), options.Applications[0])];
        await Assert.ThrowsAsync<InvalidDataException>( () =>
Engine().CycleAsync(default));
        Assert.Equal(0, machine.InstallCalls);
    }
    /// <summary>
    /// Verifies that removed assignment does not run queued work from cache.
    /// </summary>
    [Fact]
    public async Task RemovedAssignment_DoesNotRunQueuedWorkFromCache()
    {
        options.Applications = [];
        await Engine().CycleAsync(default);
        Assert.Equal(0, machine.InstallCalls);
        Assert.Null(api.Terminal);
    }
    /// <summary>
    /// Verifies that desktop install closes before MSI and succeeds without running service.
    /// </summary>
    [Fact]
    public async Task DesktopInstall_ClosesBeforeMsiAndSucceedsWithoutRunningService()
    {
        options.Applications[0].Kind = "Desktop";
        options.Applications[0].ServiceName = "";
        options.Applications[0].ExecutablePath = Path.Combine(directory, "widget.exe");
        await Engine().CycleAsync(default);
        Assert.Equal("Succeeded", api.Terminal!.Status);
        Assert.True(machine.PreparedBeforeInstall);
        await Engine().CycleAsync(default);
        Assert.Equal("Ready", api.Reports.Last().Status);
    }
    /// <summary>
    /// Verifies that desktop close failure does not start installer.
    /// </summary>
    [Fact]
    public async Task DesktopCloseFailure_DoesNotStartInstaller()

```

```

{
    options.Applications[0].Kind = "Desktop";
    options.Applications[0].ServiceName = "";
    options.Applications[0].ExecutablePath = Path.Combine(directory, "widget.exe");
    machine.FailPreparation = true;
    await Engine().CycleAsync(default);
    Assert.Equal("Failed", api.Terminal!.Status);
    Assert.Equal(0, machine.InstallCalls);
    Assert.Equal(0, machine.ConfigureCalls);
}
/// <summary>
/// Verifies that install then reports fresh matching observation.
/// </summary>
[Fact]
public async Task Install_ThenReportsFreshMatchingObservation()
{
    await Engine().CycleAsync(default);
    Assert.Equal(1, machine.InstallCalls);
    Assert.Equal(1, machine.ConfigureCalls);
    Assert.Equal("Succeeded", api.Terminal!.Status);
    Assert.Null(store.Value.Execution);
    await Engine().CycleAsync(default);
    StateReport report = api.Reports.Last();
    Assert.Equal("Running", report.Status);
    Assert.Equal(api.State.ProductVersionId, report.InstalledProductVersionId);
    Assert.Equal(api.State.EffectiveConfigurationRevisionId,
report.EffectiveConfigurationRevisionId);
    Assert.Equal(FakeApi.Hash, report.ConfigurationHash);
    Assert.Equal(1, machine.InstallCalls);
}
/// <summary>
/// Verifies that cannot launch installer without durable intent.
/// </summary>
[Fact]
public async Task CannotLaunchInstallerWithoutDurableIntent()
{
    store.BeforeSave = state =>
    {
        if (state.Execution?.Phase == "Installing") { throw new
IOException("Simulated disk failure."); }
    };
    await Assert.ThrowsAsync<IOException>(() => Engine().CycleAsync(default));
    Assert.Equal(0, machine.InstallCalls);
    Assert.Null(api.Terminal);
}
/// <summary>
/// Verifies that successful exit without installed identity does not claim success.
/// </summary>
[Fact]
public async Task SuccessfulExitWithoutInstalledIdentity_DoesNotClaimSuccess()
{
    machine.DoNotInstall = true;
    await Engine().CycleAsync(default);
    Assert.Equal("Failed", api.Terminal!.Status);
    Assert.Equal("MSI_INSTALL_FAILED", api.Terminal!.ErrorCode);
    Assert.Equal(0, machine.ConfigureCalls);
}
/// <summary>
/// Verifies that configuration failure preserves separate failure code.
/// </summary>
[Fact]
public async Task ConfigurationFailure_PreservesSeparateFailureCode()

```

```

{
    machine.FailConfiguration = true;
    await Engine().CycleAsync(default);
    Assert.Equal(1, machine.InstallCalls);
    Assert.Equal("CONFIGURATION_APPLY_FAILED", api.Terminal!.ErrorCode);
    Assert.Null(store.Value.Installations.Single().Applied);
}
/// <summary>
/// Verifies that recovery after recorded reboot does not require another reboot.
/// </summary>
[Fact]
public async Task RecoveryAfterRecordedReboot_DoesNotRequireAnotherReboot()
{
    store.Value.Execution = new()
    {
        Deployment = api.Deployment, Target = api.Target, Attempt = api.Attempt,
        Phase = "Installed", RestartRequired = true, RestartFrom = "previous-boot"
    };
    machine.InstalledVersion = api.State.ProductVersionId;
    await Engine().CycleAsync(default);
    Assert.Equal(0, machine.InstallCalls);
    Assert.False(machine.ConfigurationDeferred);
    Assert.Equal("Succeeded", api.Terminal!.Status);
}
/// <summary>
/// Verifies that configure does not invoke installer.
/// </summary>
[Fact]
public async Task Configure_DoesNotInvokeInstaller()
{
    api.Operation = "Configure";
    machine.InstalledVersion = api.State.ProductVersionId;
    await Engine().CycleAsync(default);
    Assert.Equal(0, machine.InstallCalls);
    Assert.Equal(1, machine.ConfigureCalls);
    Assert.Equal("Succeeded", api.Terminal!.Status);
}
/// <summary>
/// Verifies that configure rejects missing application.
/// </summary>
[Fact]
public async Task Configure_RejectsMissingApplication()
{
    api.Operation = "Configure";
    await Engine().CycleAsync(default);
    Assert.Equal(0, machine.InstallCalls);
    Assert.Equal(0, machine.ConfigureCalls);
    Assert.Equal("Failed", api.Terminal!.Status);
}
/// <summary>
/// Verifies that update installs mapped version.
/// </summary>
[Fact]
public async Task Update_InstallsMappedVersion()
{
    api.Operation = "Update";
    machine.InstalledVersion = Guid.NewGuid();
    await Engine().CycleAsync(default);
    Assert.Equal(1, machine.InstallCalls);
    Assert.Equal(api.State.ProductVersionId, machine.InstalledVersion);
}
/// <summary>

```

```

/// Verifies that installer failure does not apply configuration.
/// </summary>
[Fact]
public async Task InstallerFailure_DoesNotApplyConfiguration()
{
    machine.ExitCode = 1603;
    await Engine().CycleAsync(default);
    Assert.Equal("MSI_INSTALL_FAILED", api.Terminal!.ErrorCode);
    Assert.Equal(0, machine.ConfigureCalls);
    Assert.Null(store.Value.Installations.Single().Applied);
}
/// <summary>
/// Verifies that reboot required prepares configuration and waits for new boot.
/// </summary>
[Fact]
public async Task RebootRequired_PreparesConfigurationAndWaitsForNewBoot()
{
    machine.ExitCode = 3010;
    await Engine().CycleAsync(default);
    Assert.Equal("RestartRequired", api.Terminal!.Status);
    Assert.True(machine.ConfigurationDeferred);
    await Engine().CycleAsync(default);
    Assert.Equal("RestartRequired", api.Reports.Last().Status);
    machine.BootId = "new-boot";
    await Engine().CycleAsync(default);
    Assert.Equal("Running", api.Reports.Last().Status);
    Assert.Equal(1, machine.InstallCalls);
}
/// <summary>
/// Verifies that corrupt download cannot reach installer.
/// </summary>
[Fact]
public async Task CorruptDownload_CannotReachInstaller()
{
    api.CorruptDownload = true;
    await Engine().CycleAsync(default);
    Assert.Equal("DOWNLOAD_FAILED", api.Terminal!.ErrorCode);
    Assert.Equal(0, machine.InstallCalls);
}
/// <summary>
/// Verifies that superseded after download prevents local execution.
/// </summary>
[Fact]
public async Task SupersededAfterDownload_PreventsLocalExecution()
{
    api.ChangeAfterDownload = true;
    await Engine().CycleAsync(default);
    Assert.Equal("INTERRUPTED", api.Terminal!.ErrorCode);
    Assert.Equal(0, machine.InstallCalls);
    Assert.Equal(0, machine.ConfigureCalls);
}
/// <summary>
/// Verifies that lost terminal response is replayed exactly after restart.
/// </summary>
[Fact]
public async Task LostTerminalResponse_IsReplayedExactlyAfterRestart()
{
    api.LoseTerminalResponse = true;
    await Assert.ThrowsAsync<HttpRequestException>(() =>
Engine().CycleAsync(default));
    AttemptResult original = store.Value.Execution!.Result!;
    await Engine().CycleAsync(default);

```

```

        Assert.Equal(original, api.Terminal);
        Assert.Equal(2, api.TerminalRequests.Count);
        Assert.All(api.TerminalRequests, result => Assert.Equal(original, result));
        Assert.Equal(1, machine.InstallCalls);
        Assert.Equal(1, machine.ConfigureCalls);
    }
    /// <summary>
    /// Verifies that lost start response recovers original attempt before execution.
    /// </summary>
    [Fact]
    public async Task LostStartResponse_RecoversOriginalAttemptBeforeExecution()
    {
        api.LoseStartResponse = true;
        await Assert.ThrowsAsync<HttpRequestException>(() =>
Engine().CycleAsync(default));
        Assert.NotNull(store.Value.Execution);
        Assert.Null(store.Value.Execution.Attempt);
        await Engine().CycleAsync(default);
        Assert.Equal(1, machine.InstallCalls);
        Assert.Equal("Succeeded", api.Terminal!.Status);
    }
    /// <summary>
    /// Verifies that restart during installer waits then reports interrupted without
    relaunch.
    /// </summary>
    [Fact]
    public async Task RestartDuringInstaller_WaitsThenReportsInterruptedWithoutRelaunch()
    {
        store.Value.Execution = new() { Deployment = api.Deployment, Target = api.Target,
Attempt = api.Attempt, Phase = "Installing" };
        machine.Busy = true;
        await Engine().CycleAsync(default);
        Assert.Null(api.Terminal);
        Assert.Equal(0, machine.InstallCalls);
        machine.Busy = false;
        await Engine().CycleAsync(default);
        Assert.Equal("INTERRUPTED", api.Terminal!.ErrorCode);
        Assert.Equal(0, machine.InstallCalls);
    }
    /// <summary>
    /// Verifies that restart after installation resumes only configuration.
    /// </summary>
    [Fact]
    public async Task RestartAfterInstallation_ResumesOnlyConfiguration()
    {
        store.Value.Execution = new() { Deployment = api.Deployment, Target = api.Target,
Attempt = api.Attempt, Phase = "Installed" };
        machine.InstalledVersion = api.State.ProductVersionId;
        await Engine().CycleAsync(default);
        Assert.Equal(0, machine.InstallCalls);
        Assert.Equal(1, machine.ConfigureCalls);
        Assert.Equal("Succeeded", api.Terminal!.Status);
    }
    /// <summary>
    /// Verifies that recovery cannot authorize an unknown attempt after supersession.
    /// </summary>
    [Fact]
    public async Task RecoveryCannotAuthorizeAnUnknownAttemptAfterSupersession()
    {
        store.Value.Execution = new() { Deployment = api.Deployment, Target = api.Target
};
        api.RejectStart = true;

```

```

        await Assert.ThrowsAsync<AgentApiException>(() => Engine().CycleAsync(default));
        Assert.NotNull(store.Value.Execution);
        Assert.Equal(0, machine.InstallCalls);
    }
    /// <summary>
    /// Verifies that unknown version mapping does not reserve work.
    /// </summary>
    [Fact]
    public async Task UnknownVersionMapping_DoesNotReserveWork()
    {
        options.Applications[0].Versions = [new(Guid.NewGuid(), Guid.NewGuid(), "2.0.0")];
        await Engine().CycleAsync(default);
        Assert.Equal(0, api.StartCalls);
        Assert.Equal(0, machine.InstallCalls);
    }
    /// <summary>
    /// Verifies that reports retain monotonic timestamps after restart and clock change.
    /// </summary>
    [Fact]
    public async Task ReportsRetainMonotonicTimestampsAfterRestartAndClockChange()
    {
        api.NoWork = true;
        await Engine().CycleAsync(default);
        DateTimeOffset first = api.Reports.Last().ReportedAt;
        api.UtcNow = api.UtcNow.AddSeconds(-10);
        await Engine().CycleAsync(default);
        Assert.True(api.Reports.Last().ReportedAt > first);
    }
    /// <summary>
    /// Verifies that lost state response replays saved payload before new observation.
    /// </summary>
    [Fact]
    public async Task LostStateResponse_ReplaysSavedPayloadBeforeNewObservation()
    {
        api.NoWork = true;
        api.LoseReportResponse = true;
        await Assert.ThrowsAsync<HttpRequestException>(() =>
            Engine().CycleAsync(default));
        StateReport original = store.Value.Installations[0].PendingReport!;
        machine.InstalledVersion = Guid.NewGuid();
        await Engine().CycleAsync(default);
        Assert.Equal(original, api.Reports[0]);
        Assert.Equal(original, api.Reports[1]);
        Assert.NotEqual(original, api.Reports[2]);
    }
    /// <summary>
    /// Releases the resources owned by this instance.
    /// </summary>
    public void Dispose() { TestDirectory.Delete(directory, "BridgeControlAgentTests"); }

    /// <summary>
    /// Simulates durable agent state by storing independent in-memory copies.
    /// </summary>
    internal sealed class MemoryStore : IStateStore
    {
        public AgentState Value { get; set; } = new();
        public Action<AgentState>? BeforeSave { get; set; }
        /// <summary>
        /// Returns an independent copy of the saved test agent state.
        /// </summary>
        public AgentState Load() => Clone(Value);
        /// <summary>

```

```

        /// Stores an independent copy of the test agent state.
        /// </summary>
        public void Save(AgentState state) { BeforeSave?.Invoke(state); Value =
Clone(state); }
        /// <summary>
        /// Copies agent state through JSON serialization to simulate persistence
boundaries.
        /// </summary>
        private static AgentState Clone(AgentState value) =>
JsonSerializer.Deserialize<AgentState>(JsonSerializer.Serialize(value));
    }
    /// <summary>
    /// Simulates agent API responses and records deployment interactions for tests.
    /// </summary>
    internal sealed class FakeApi : IAgentApi
    {
        public IReadOnlyList<Assignment> Assignments { get; set; } = [];
        public int RegistrationCalls { get; private set; }
        /// <summary>
        /// Returns the configured application assignments for the test agent.
        /// </summary>
        public Task<IReadOnlyList<Assignment>> AssignmentsAsync(CancellationTokens ct) =>
Task.FromResult(Assignments);
        public DateTimeOffset.UtcNow { get; set; } = DateTimeOffset.UtcNow;
        public Guid AgentId { get; } = Guid.NewGuid();
        public DesiredState State { get; set; } = new(Guid.NewGuid(), Guid.NewGuid(),
Guid.NewGuid(), DateTimeOffset.UtcNow);
        public string Operation { get; set; } = "Install";
        public Target Target => new(State, "/api/v1/agent/artifacts/installer/content",
"/api/v1/agent/artifacts/config/content");
        public Deployment Deployment => new(DeploymentId, State.InstallationId,
State.ProductVersionId, State.EffectiveConfigurationRevisionId, Operation);
        public Guid DeploymentId { get; } = Guid.NewGuid();
        public Attempt Attempt { get; } = new(Guid.NewGuid(), 1);
        public static string Hash => Convert.ToHexStringLower(SHA256.HashData("{}"u8));
        public List<StateReport> Reports { get; } = [];
        public List<AttemptResult> TerminalRequests { get; } = [];
        public AttemptResult? Terminal { get; set; }
        public bool CorruptDownload { get; set; }
        public bool ChangeAfterDownload { get; set; }
        public bool LoseTerminalResponse { get; set; }
        public bool LoseStartResponse { get; set; }
        public bool LoseReportResponse { get; set; }
        public bool RejectStart { get; set; }
        public bool NoWork { get; set; }
        public int StartCalls { get; set; }
        /// <summary>
        /// Returns a simulated installation registration.
        /// </summary>
        public Task<Installation> RegisterInstallationAsync(ApplicationOptions app,
CancellationTokens ct)
        {
            RegistrationCalls++;
            return Task.FromResult(new Installation(State.InstallationId,
app.ProductId, AgentId, app.Name));
        }
        /// <summary>
        /// Accepts a simulated agent heartbeat.
        /// </summary>
        public Task HeartbeatAsync(Heartbeat heartbeat, CancellationTokens ct) =>
Task.CompletedTask;
        /// <summary>

```

```

        /// Accepts a simulated operational log entry.
        /// </summary>
        public Task LogAsync(Guid installationId, string message, CancellationToken ct) =>
Task.CompletedTask;
        /// <summary>
        /// Records the simulated installation-state report and configured response
failure.
        /// </summary>
        public Task ReportAsync(Guid installationId, StateReport report, CancellationToken
ct)
        {
            Reports.Add(report);
            if (LoseReportResponse) { LoseReportResponse = false; throw new
HttpRequestException(); }
            return Task.CompletedTask;
        }
        /// <summary>
        /// Returns the configured desired target for the test installation.
        /// </summary>
        public Task<Target> DesiredAsync(Guid installationId, CancellationToken ct) =>
Task.FromResult(Target);
        /// <summary>
        /// Returns the configured pending deployment work.
        /// </summary>
        public Task<IReadOnlyList<Deployment>> PendingAsync(CancellationToken ct) =>
Task.FromResult<IReadOnlyList<Deployment>>(NoWork || Terminal is not null ? [] : [Deployment]);
        /// <summary>
        /// Records a simulated attempt start and returns its configured response.
        /// </summary>
        public Task<Attempt> StartAsync(Guid deploymentId, CancellationToken ct)
        {
            StartCalls++;
            if (LoseStartResponse) { LoseStartResponse = false; throw new
HttpRequestException(); }
            if (RejectStart) { throw new AgentApiException(HttpStatusCode.Conflict); }
            return Task.FromResult(Aattempt);
        }
        /// <summary>
        /// Records a simulated attempt result and applies configured response failures.
        /// </summary>
        public Task UpdateAsync(Guid deploymentId, Guid attemptId, AttemptResult result,
CancellationToken ct)
        {
            Assert.Equal(Aattempt.AttemptId, attemptId);
            if (result.CompletedAt is not null)
            {
                TerminalRequests.Add(result);
                if (Terminal is not null) { Assert.Equal(Terminal, result); }
                Terminal = result;
                if (LoseTerminalResponse) { LoseTerminalResponse = false; throw new
HttpRequestException(); }
            }
            return Task.CompletedTask;
        }
        /// <summary>
        /// Writes the simulated artifact content for a test download.
        /// </summary>
        public Task<Download> DownloadAsync(string url, DesiredState expected, string
path, CancellationToken ct)
        {
            if (CorruptDownload) { throw new InvalidDataException(); }
            File.WriteAllText(path, "{}");

```

```

        if (ChangeAfterDownload) { State = State with {
EffectiveConfigurationRevisionId = Guid.NewGuid() }; }
        return Task.FromResult(new Download(path, Hash));
    }
}
/// <summary>
/// Simulates installed application state and local deployment operations.
/// </summary>
internal sealed class FakeMachine : IManagedApplication
{
    public string BootId { get; set; } = "first-boot";
    public bool Busy { get; set; }
    public Guid TargetVersion { get; set; }
    public Guid? InstalledVersion { get; set; }
    public int ExitCode { get; set; }
    public int InstallCalls { get; set; }
    public int ConfigureCalls { get; set; }
    public bool ConfigurationDeferred { get; set; }
    public bool DoNotInstall { get; set; }
    public bool FailConfiguration { get; set; }
    public bool FailPreparation { get; set; }
    public bool PreparedBeforeInstall { get; set; }
    /// <summary>
    /// Records preparation for a simulated application update.
    /// </summary>
    public Task PrepareUpdateAsync(ApplicationOptions app, CancellationToken ct)
    {
        if (FailPreparation) { throw new IOException("Unable to close desktop."); }
        PreparedBeforeInstall = InstallCalls == 0;
        return Task.CompletedTask;
    }
    /// <summary>
    /// Returns the simulated installer activity state.
    /// </summary>
    public bool InstallerIsRunning() => Busy;
    /// <summary>
    /// Returns the simulated installed application evidence.
    /// </summary>
    public bool ResolveMsiIdentity { get; set; }
    public Observation Observe(ApplicationOptions app, AppliedConfiguration? applied)
    {
        if (InstalledVersion is null) { return new(null, null, null,
"NotInstalled"); }
        Guid? version = InstalledVersion;
        if (ResolveMsiIdentity)
        {
            VersionBinding msi = app.Versions.Single(binding =>
binding.ProductVersionId == InstalledVersion);
            version = InstalledVersionResolver.Resolve(app, msi.ProductCode,
msi.Version, applied)?.ProductVersionId;
            if (version is null) { return new(null, null, null, "Unknown"); }
        }
        return new(version, applied?.RevisionId, applied?.Hash, applied?.RebootFrom
== BootId ? "RestartRequired" : app.Kind == "Desktop" ? "Ready" : "Running");
    }
    /// <summary>
    /// Simulates installer validation for the test application.
    /// </summary>
    public void ValidateInstaller(ApplicationOptions app, VersionBinding version,
string path) { }
    /// <summary>
    /// Records a simulated installer launch and returns the configured exit code.

```

```

        /// </summary>
        public Task<int> InstallAsync(string path, CancellationToken ct)
        {
            InstallCalls++;
            if (!DoNotInstall && ExitCode is 0 or 3010) { InstalledVersion =
TargetVersion; }
            return Task.FromResult(ExitCode);
        }
        /// <summary>
        /// Records simulated configuration application and any configured failure.
        /// </summary>
        public Task ApplyConfigurationAsync(ApplicationOptions app, string path, bool
restartRequired, CancellationToken ct)
        {
            ConfigureCalls++;
            if (FailConfiguration) { throw new IOException("Simulated configuration
failure."); }
            ConfigurationDeferred = restartRequired;
            return Task.CompletedTask;
        }
    }
}

```

DesktopApplicationTests.cs

```

using BridgeControl.Agent;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Verifies desktop application behavior and regression scenarios.
/// </summary>
public sealed class DesktopApplicationTests
{
    /// <summary>
    /// Verifies that desktop configuration writes without service and close failure preserves
file.
    /// </summary>
    [Theory]
    [InlineData(false)]
    [InlineData(true)]
    public async Task
DesktopConfiguration_WritesWithoutService_AndCloseFailurePreservesFile(bool failClose)
    {
        string directory = Path.Combine(Path.GetTempPath(), "BridgeControlDesktopTests",
Guid.NewGuid().ToString("N"));
        Directory.CreateDirectory(directory);
        try
        {
            string source = Path.Combine(directory, "download.json"), destination =
Path.Combine(directory, "settings.json");
            await File.WriteAllTextAsync(source, "{\"updated\":true}");
            await File.WriteAllTextAsync(destination, "{\"original\":true}");
            var processes = new FakeDesktop { Fail = failClose };
            var machine = new WindowsApplication(processes);
            var app = new ApplicationOptions { Kind = "Desktop", ExecutablePath =
@"C:\Program Files\Widget\widget.exe", ConfigurationPath = destination };

```

```

        if (failClose)
        {
            await Assert.ThrowsAsync<IOException>(() =>
machine.ApplyConfigurationAsync(app, source, false, default));
            Assert.Equal("{\"original\":true}", await
File.ReadAllTextAsync(destination));
        }
        else
        {
            await machine.ApplyConfigurationAsync(app, source, false, default);
            Assert.Equal(await File.ReadAllTextAsync(source), await
File.ReadAllTextAsync(destination));
        }
        Assert.Equal(app.ExecutablePath, processes.ClosedPath);
    }
    finally { Directory.Delete(directory, true); }
}
/// <summary>
/// Verifies that desktop options require exact executable but no service and reject
duplicate executable.
/// </summary>
[Fact]
public void
DesktopOptions_RequireExactExecutableButNoService_AndRejectDuplicateExecutable()
{
    var app = new ApplicationOptions
    {
        Kind = "Desktop", ProductId = Guid.NewGuid(), UpgradeCode = Guid.NewGuid(),
Name = "Widget",
        ExecutablePath = @"C:\Program Files\Widget\widget.exe", ConfigurationPath =
@"C:\ProgramData\Widget\config.json",
        Versions = [new(Guid.NewGuid(), Guid.NewGuid(), "1.0.0")]
    };
    var options = new AgentOptions { ApiBaseUrl = "https://localhost/", Applications =
[app] };
    options.Validate();
    app.ExecutablePath = "widget.exe";
    Assert.Throws<InvalidOperationException>(() => options.Validate());
    app.ExecutablePath = @"C:\Program Files\Widget\widget.exe";
    app.ServiceName = "InvalidForDesktop";
    Assert.Throws<InvalidOperationException>(() => options.Validate());
    app.ServiceName = "";
    options.Applications.Add(new()
    {
        Kind = "Desktop", ProductId = Guid.NewGuid(), UpgradeCode = Guid.NewGuid(),
Name = "Other",
        ExecutablePath = app.ExecutablePath.ToUpperInvariant(), ConfigurationPath =
@"C:\ProgramData\Widget\other.json", Versions = app.Versions
    });
    Assert.Throws<InvalidOperationException>(() => options.Validate());
}
/// <summary>
/// Verifies that reconciliation ready requires matching evidence stopped services still
need activation.
/// </summary>
[Theory]
[InlineData("Ready", true, "Matching")]
[InlineData("Ready", false, "Configure")]
[InlineData("Stopped", true, "Configure")]
[InlineData("RestartRequired", true, "Configure")]

```

```

        public void
Reconciliation_ReadyRequiresMatchingEvidence_StoppedServicesStillNeedActivation(string status,
bool matches, string expected)
    {
        var target = new StateTarget(new() { Id = Guid.NewGuid() }, new() { Id =
Guid.NewGuid(), FileArtifact = new() { Sha256 = new string('a', 64) } }, new());
        var report = new InstallationStateReport
        {
            Status = status, ReportedAt = DateTimeOffset.UtcNow,
InstalledProductVersionId = target.Version.Id,
            EffectiveConfigurationRevisionId = target.Configuration.Id,
ConfigurationHash = matches ? target.Configuration.FileArtifact.Sha256 : new string('b', 64)
        };
        Assert.Equal(expected, StateReconciliationProcessor.Classify(report, target,
DateTimeOffset.UtcNow));
    }
    /// <summary>
    /// Simulates desktop process shutdown for application update tests.
    /// </summary>
    private sealed class FakeDesktop : IDesktopProcesses
    {
        public bool Fail { get; init; }
        public string? ClosedPath { get; private set; }
        /// <summary>
        /// Records a simulated desktop shutdown or returns the configured failure.
        /// </summary>
        public Task CloseAsync(string executablePath, CancellationToken ct)
        {
            ClosedPath = executablePath;
            if (Fail) { throw new IOException("Unable to close application."); }
            return Task.CompletedTask;
        }
    }
}

```

InstalledVersionResolverTests.cs

```

using BridgeControl.Agent;

namespace BridgeControl.Agent.Tests;

/// <summary>Verifies duplicate release mappings preserve observed installation
identity.</summary>
public sealed class InstalledVersionResolverTests
{
    private readonly Guid oldCode = Guid.NewGuid();
    private readonly Guid newCode = Guid.NewGuid();
    private ApplicationOptions Application() => new()
    {
        ProductId = Guid.NewGuid(), UpgradeCode = Guid.NewGuid(), Name = "Widget", Kind =
"Desktop",
        ConfigurationPath = @"C:\Program Files\Widget\config.json", ExecutablePath =
@"C:\Program Files\Widget\Widget.exe",
        Versions = [new(Guid.NewGuid(), oldCode, "1.0.0"), new(Guid.NewGuid(), oldCode,
"1.0.0"), new(Guid.NewGuid(), newCode, "1.0.2")]
    };
    private static AgentOptions Options(ApplicationOptions app) => new() { ApiBaseUrl =
"https://bridge.example/", Applications = [app] };

    [Fact]

```

```

    public void DuplicateMsiIdentity_UsesRecordedRelease_RegardlessOfOrderOrDesiredVersion()
    {
        ApplicationOptions app = Application();
        Options(app).Validate();
        VersionBinding original = app.Versions[0];
        app.SettingsVersionId = app.Versions[1].ProductVersionId;
        AppliedConfiguration applied = new(original.ProductVersionId, Guid.NewGuid(),
"hash", null);
        Assert.Equal(original, InstalledVersionResolver.Resolve(app, oldCode, "1.0.0",
applied));
        app.Versions.Reverse();
        Assert.Equal(original, InstalledVersionResolver.Resolve(app, oldCode, "1.0.0",
applied));
        Assert.Equal(app.Versions[0], InstalledVersionResolver.Resolve(app, newCode,
"1.0.2", applied));
    }

    [Fact]
    public void AmbiguousOrUnmatchedMsi_DoesNotInventAnInstalledVersion()
    {
        ApplicationOptions app = Application();
        app.SettingsVersionId = app.Versions[0].ProductVersionId;
        Assert.Null(InstalledVersionResolver.Resolve(app, oldCode, "1.0.0", null));
        AppliedConfiguration unrelated = new(app.Versions[2].ProductVersionId,
Guid.NewGuid(), "hash", null);
        Assert.Null(InstalledVersionResolver.Resolve(app, oldCode, "1.0.0", unrelated));
        Assert.Null(InstalledVersionResolver.Resolve(app, oldCode, "1.0.2", unrelated));
        Assert.Null(InstalledVersionResolver.Resolve(app, Guid.NewGuid(), "1.0.0",
unrelated));
    }

    [Fact]
    public void ConflictingVersionsOrDuplicateApiIds_AreStillRejected()
    {
        ApplicationOptions app = Application();
        app.Versions[1] = app.Versions[1] with { Version = "1.0.1" };
        Assert.Throws<InvalidOperationException>(() => Options(app).Validate());
        app.Versions[1] = app.Versions[0];
        Assert.Throws<InvalidOperationException>(() => Options(app).Validate());
    }
}

```

InstallerActivityTests.cs

```

using BridgeControl.Agent;

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Verifies installer activity behavior and regression scenarios.
/// </summary>
public sealed class InstallerActivityTests
{
    /// <summary>
    /// Verifies that missing or idle installer mutex does not block observation.
    /// </summary>
    [Fact]
    public void MissingOrIdleInstallerMutex_DoesNotBlockObservation()
    {
        string name = @"Local\BridgeControlTest_" + Guid.NewGuid().ToString("N");
    }
}

```

```

        Assert.False(InstallerActivity.IsBusy(name));
        using Mutex mutex = new(false, name);
        Assert.False(InstallerActivity.IsBusy(name));
        // The probe must release the mutex it briefly acquires.
        bool available = false;
        Thread thread = new(() =>
        {
            available = mutex.WaitOne(0);
            if (available) { mutex.ReleaseMutex(); }
        });
        thread.Start();
        Assert.True(thread.Join(TimeSpan.FromSeconds(5)));
        Assert.True(available);
    }

    /// <summary>
    /// Verifies that active installer blocks until release or abandonment.
    /// </summary>
    [Theory]
    [InlineData(false)]
    [InlineData(true)]
    public void ActiveInstallerBlocksUntilReleaseOrAbandonment(bool abandon)
    {
        string name = @"Local\BridgeControlTest_" + Guid.NewGuid().ToString("N");
        using Mutex mutex = new(false, name);
        using ManualResetEventSlim held = new();
        using ManualResetEventSlim finish = new();
        Thread owner = new(() =>
        {
            mutex.WaitOne();
            held.Set();
            finish.Wait();
            if (!abandon) { mutex.ReleaseMutex(); }
        });
        owner.Start();
        try
        {
            Assert.True(held.Wait(TimeSpan.FromSeconds(5)));
            Assert.True(InstallerActivity.IsBusy(name));
        }
        finally { finish.Set(); owner.Join(); }
        Assert.False(InstallerActivity.IsBusy(name));
        Assert.False(InstallerActivity.IsBusy(name));
    }
}

```

TestDirectory.cs

```

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Removes temporary test directories with retry handling for delayed file releases.
/// </summary>
internal static class TestDirectory
{
    /// <summary>
    /// Deletes a temporary test directory, retrying transient file access failures.
    /// </summary>
    public static void Delete(string directory, string parentName)
    {

```

```

        string fullPath = Path.GetFullPath(directory);
        string parent = Path.GetFullPath(Path.Combine(Path.GetTempPath(), parentName));
        if (!string.Equals(Path.GetDirectoryName(fullPath), parent,
            StringComparison.OrdinalIgnoreCase)
            || !Guid.TryParseExact(Path.GetFileName(fullPath), "N", out _))
        {
            throw new InvalidOperationException("Refusing to remove a directory outside
this test's generated temporary storage.");
        }
        Directory.Delete(fullPath, true);
    }
}

```

WindowsApplicationTests.cs

```

using System.Runtime.InteropServices;
using BridgeControl.Agent;

namespace BridgeControl.Agent.Tests;

/// <summary>
/// Verifies windows application behavior and regression scenarios.
/// </summary>
public sealed class WindowsApplicationTests
{
    /// <summary>
    /// Verifies that native observation reports absent product without inventing version.
    /// </summary>
    [Fact]
    public void NativeObservation_ReportsAbsentProductWithoutInventingVersion()
    {
        WindowsApplication application = new();
        Assert.False(string.IsNullOrEmpty(application.BootId));
        Observation observed = application.Observe(new() { UpgradeCode = Guid.NewGuid() },
null);
        Assert.Equal(new Observation(null, null, null, "NotInstalled"), observed);
    }
    /// <summary>
    /// Verifies that native installer inspection requires exact product version and upgrade
identity.
    /// </summary>
    [Fact]
    public void NativeInstallerInspection_RequiresExactProductVersionAndUpgradeIdentity()
    {
        string path = Path.Combine(Path.GetTempPath(), "BridgeControlAgent_" +
Guid.NewGuid().ToString("N") + ".msi");
        Guid upgrade = Guid.NewGuid(), product = Guid.NewGuid();
        uint database = 0;
        try
        {
            Assert.Equal(0u, MsiOpenDatabase(path, new IntPtr(3), out database));
            Execute(database, "CREATE TABLE `Property` (`Property` CHAR(72) NOT NULL,
`Value` CHAR(0) LOCALIZABLE PRIMARY KEY `Property`)");
            Execute(database, $"INSERT INTO `Property` (`Property`, `Value`) VALUES
('UpgradeCode', '{upgrade:B}')");
            Execute(database, $"INSERT INTO `Property` (`Property`, `Value`) VALUES
('ProductCode', '{product:B}')");
            Execute(database, "INSERT INTO `Property` (`Property`, `Value`) VALUES
('ProductVersion', '1.2.3')");
            Assert.Equal(0u, MsiDatabaseCommit(database));
        }
        catch { }
    }
}

```

```

        MsiCloseHandle(database);
        database = 0;
        WindowsApplication application = new();
        ApplicationOptions options = new() { UpgradeCode = upgrade };
        VersionBinding version = new(Guid.NewGuid(), product, "1.2.3");
        application.ValidateInstaller(options, version, path);
        Assert.Throws<InvalidDataException>(() =>
application.ValidateInstaller(options, version with { Version = "2.0.0" }, path));
        Assert.Throws<InvalidDataException>(() =>
application.ValidateInstaller(options, version with { ProductCode = Guid.NewGuid() }, path));
        Assert.Throws<InvalidDataException>(() =>
application.ValidateInstaller(new() { UpgradeCode = Guid.NewGuid() }, version, path));
    }
    finally
    {
        if (database != 0) { MsiCloseHandle(database); }
        if (File.Exists(path)) { File.Delete(path); }
    }
}
/// <summary>
/// Executes a SQL statement against the test MSI database.
/// </summary>
private static void Execute(uint database, string query)
{
    Assert.Equal(0u, MsiDatabaseOpenView(database, query, out uint view));
    try { Assert.Equal(0u, MsiViewExecute(view, 0)); }
    finally { MsiCloseHandle(view); }
}
/// <summary>
/// Opens a Windows Installer database through the native MSI API.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiOpenDatabaseW")] private
static extern uint MsiOpenDatabase(string path, IntPtr persist, out uint database);
/// <summary>
/// Creates a native MSI database view for a SQL query.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiDatabaseOpenViewW")]
private static extern uint MsiDatabaseOpenView(uint database, string query, out uint view);
/// <summary>
/// Executes a native MSI database view with the supplied record.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiViewExecute(uint view, uint record);
/// <summary>
/// Commits changes to a native MSI database.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiDatabaseCommit(uint database);
/// <summary>
/// Releases a native Windows Installer handle.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiCloseHandle(uint handle);
}

```